Intermediate distance sampling workshop - St Andrews 2017

Laura Marshall, David L. Miller and Len Thomas

2017-06-28

Contents

Preface

A bookdown version of the workshop practicals. Section numbers correspond to practicals and topics:

Section number	Topic	Scheduled
1	R tutorial	Sunday pm
2	simple ds() of simulated data	Monday am
3	awkward data	Monday pm
4	DSsim	Monday pm
5	Sperm whale intro	Tuesday am
6	density surface	Tuesday am
7	density surface II	Tuesday pm
8	prediction with dsm	Wednesday am
9	variance with dsm	Wednesday pm
10	double observer	Thursday am

Chapter 1

A hands on introduction to R tutorial

prepared by Tiago A. Marques, Danielle Harris & Len Thomas

1.1 Introduction

This tutorial was created as a gentle introduction to the R environment. It does not assume any basic knowledge about R, but some basic programming notions would be desirable.

There is an extensive community revolving around R, and abundant courses, tutorials, books, blogs, list servers, etc, freely available online. We provide here a small list of some of these:

- R webpage - the main R webpage, including links to downloading R, manuals, tutorials, dedicated search engines, etc.
- R video tutorials - video how to's in R
- Online tutorial - a course with interactive exercises
- Online course - DataCamp commercial site
- Reference card - A very handy list of useful R functions
- Short reference card - A longer reference card with most commonly used R functions

To facilitate the interaction with R we leverage on RStudio, a piece of software which allows users to have at a click's distance many useful features in R. In the following sections of the tutorial you will be guided through a first session of R via RStudio.

The tutorial is intended to follow a brief presentation about R and RStudio, their interaction and capabilities. It assumes that R and RStudio have been previously installed in the computer you are using. The latest version of both software packages is recommended. Both are free and open source.

1.2 Introduction to RStudio

Most users (except perhaps die-hard command line users) will use some sort of graphical user interface (GUI) to R. While the basic R installation comes with a simple GUI, here we adopt the

use of RStudio, which considerably facilitates an introduction to R by providing many shortcuts and convenient features which we introduce next.

A major advantage of RStudio is that it makes it easy for you to type your R code into a script window, which you can easily save, and then send individual lines or blocks of code to the R command line to be acted upon. This way, you have a record of what you have done, in the saved script file, and can easily reproduce it any time you like. We strongly recommend that you save your code script.

Given RStudio has been installed, when you double-click on a R workspace it should open in RStudio[1]. After the presentation on R and RStudio you just sat through, from within RStudio you should be able to know where to find:

- the command line (bottom left pane[2])
- the code scripts (top left pane)
- the workspace objects (top right pane)
- the loaded packages and how to load them (bottom right pane)
- the created plots (bottom right pane)
- the help files (bottom right pane)
- a file navigator system akin to windows explorer (bottom right pane)

Note that you can customize the aspect of RStudio (e.g. font size and colours of the smart syntax highlighting scheme) via `Tools|Global options`.

A very handy feature of RStudio is that you can preview the possible arguments of functions, as well as their description, directly when you are inserting the code. Let's try doing that. Type say `seq()` in the command line or the script window and then place the cursor between the parenthesis and press the `Tab` key... Is this a nice feature or what?

Now we have *met* RStudio and we know how it can make our life simpler, let's move on.

1.3 A first session in RStudio

We have provided a R workspace named `tutorial.Rdata`. Open RStudio and then open it by selecting `File|Open File`. We recommend that you begin by creating a script file (`Ctrl+Shift+N`, RStudio Shortcut) and use that to save and comment all your code that will be executed during the tutorial. In this way you will have a record of everything you did.

You know that R is ready to receive a command when you see the R prompt on the command line (on the bottom left tab by default in RStudio): ">". If you type a line of code that is not complete, R presents the "+" character, so that the user knows it expects the conclusion of the current line. Important note: while the prompt ">" and "+" will be shown in this tutorial's code, you should not try to add either ">" nor "+" to the command line: this is something that R does for you and will complain if you try to do it yourself![3]

On the top right corner tab, where objects available in the `Environment` are listed, you can see that in `tutorial.Rdata` there are only two example objects. These are `x1` and `obj2`. We can print an object to the screen by simply typing its name and press enter (despite the fact that currently you

[1] If this fails, you might have to first associate .Rdata files with RStudio.

[2] All the tab positions are the RStudio defaults, but this can be customized by the user later.

[3] Past experience tells us that more than one person will have problems because they forgot to delete a ">" and/or "+" from the code below when they copy paste the code into their own R sessions. Avoid being that person!

can actually see the values on these objects `Environment` tab - but that is because they are simple objects and the workspace is almost empty.)

R is a very powerful calculator! Try some simple maths, say for example (you need to press enter after each line so that the line is evaluated)

```
4+3
```

```
## [1] 7
```

```
log(8)
```

```
## [1] 2.079442
```

```
sin(pi)
```

```
## [1] 1.224606e-16
```

Tip: There is actually a simpler way to do sourcing from the script file in RStudio. CTRL-Enter is a keyboard shortcut for "source the current line of code in my script file and move the cursor to the next line". In general if you like keyboard shortcuts, look in RStudio under the menu "Help | Keyboard shortcuts".

At the moment your workspace is almost empty, but we can change that easily by creating new objects. We will create a variable called `myvar1` which we will assign the value of 4. This is typically done using the assign operator `<-`.

```
myvar1 <- 4
```

There are typically multiple ways to do the same thing in R, and this is sometimes referred to as a disadvantage. For simplicity, we deliberately avoid presenting the several alternatives for each action, and concentrate on the ones we prefer. This is not the same as saying these are the best, and if you continue to work with R you will likely get used to doing things your way - for now we do it our way!

An object should have been created in your workspace. You can list all objects in a given workspace using

```
ls()
```

```
## [1] "myvar1" "obj2"   "x1"
```

You can also remove any object by using the `rm` function, so here we remove `myvar1`, `x1` and `obj2`

```
rm(myvar1,x1,obj2)
```

and hence our workspace is empty again. Note the difference between `ls()` and `rm()`. While the first function does not need any arguments, the second requires at least one argument (but can take several). This can be easily seen by checking their help files and noting that `rm()` needs at least 1 explicit argument while `ls()` can work with defaults.

```
?rm
```

This is a convenient way to obtain more information about a given function. If one does not know what the name of the function might be, one can search for functions containing a given string. The following command lists all the functions with the string "mean" in them.

```
apropos("mean")
```

```
##  [1] ".colMeans"     ".rowMeans"     "colMeans"      "kmeans"
##  [5] "mean"          "mean.Date"     "mean.default"  "mean.difftime"
##  [9] "mean.POSIXct"  "mean.POSIXlt"  "rowMeans"      "weighted.mean"
```

Not surprisingly, most if not all of these functions will be used for some kind of mean calculation. You can look into any one of them using the `?` as above. We have assigned a number to a variable , but we can actually more generally have vectors (strictly, `myvar1` was a numeric vector of length 1) containing variables. The following code assigns some numbers to 3 different vectors.

```
x2 <- c(1,2,0.12,4,-22)
x3 <- seq(1,8,by=2)
#and a useful shortcut for sequences with the by argument = 1
x1 <- 1:5
```

The function `seq` is very useful for setting sequences of numbers. The optional arguments `length.out` and `along.with` provides extra flexibility.

```
x1
```

```
## [1] 1 2 3 4 5
```

We can use the usual mathematical operators over vectors. A few examples follow:

```
x1 + x2
```

```
## [1]   2.00   4.00   3.12   8.00 -17.00
```

```
x4 <- x1 + x2
x5 <- x1 - x2
x6 <- x1 * x2
x7 <- x1 / x2
x4
```

```
## [1]   2.00   4.00   3.12   8.00 -17.00
```

```
x5
```

```
## [1]  0.00  0.00  2.88  0.00 27.00
```

```
x6
```

```
## [1]    1.00    4.00    0.36   16.00 -110.00
```

```
x7
```

```
## [1]  1.0000000  1.0000000 25.0000000  1.0000000 -0.2272727
```

Note that if the vectors are of the same length, R performs the operation element-wise. Another useful feature is that R recycles vectors if they are not the same length

```
x8 <- c(1,2,3,4)
x8 + 2
```

```
## [1] 3 4 5 6
```

However, if one of the vectors is smaller, unexpected behaviour can happen, because R recycles elements regardless (so be careful, a warning is typically produced)

```
x9 <- c(3,4,5)
x10 <- c(0.7,0.9,1.3)
x9 + x10
```

```
## [1] 3.7 4.9 6.3
```

```
x8 + x9
```

```
## Warning in x8 + x9: longer object length is not a multiple of shorter
## object length
```

```
## [1] 4 6 8 7
```

Notice that a warning message was produced when `x8` and `x9` were added. Usually these messages are important and should be read! Quite often the answer to your current question lies in the previous error or warning message.

Another useful function is `rep`, which allows one to create repetitions of patterns. As examples, see the difference between the next two lines of code

```
rep(c(1,2,3,4), times=3)
```

```
##  [1] 1 2 3 4 1 2 3 4 1 2 3 4
```

```
rep(c(1,2,3,4), each=3)
```

```
##  [1] 1 1 1 2 2 2 3 3 3 4 4 4
```

It is now time to end our first R session. At this point you need to decide what to do, as all objects created so far are in the memory, but this will be wiped out unless we explicitly save it to a file. The easiest way to do so is by calling the `save.image()` function.

```
save.image(file="my1stR.Rdata")
```

Note the unusual extension name `.Rdata` associated with R workspaces (an R file is called a workspace). We could now load up this workspace in a new R session, or typically we will load up that workspace by starting R by double clicking on the file created. Do this to see that you retrieve the above created objects. Note that if you already have an R session open, you can load up any previously saved workspace via function `load()`.

Note that you have saved your workspace in some directory but you have not defined it. By default, this is your working directory. You can check what that directory currently is by using the following command

```
getwd()
```

You can always change the directory you are working on by setting it up explicitly to your desired location, using

```
#set the working directory - but remember to use your own path!!!
setwd("C:/Users/myusername/Desktop/mycourse")
#note how you can write comments in R by using "#"
#anything in front of # is not interpreted by R
#and treated as a comment
#you should have the good habit of extensively commenting
```

```
#all your code so that you know what you've done
#when you return to it even months or years later
```

We have just started R, created and removed some objects, and used simple functions like `ls()`, `seq()` or `save()`. R is an object oriented language, and functions and vectors are just examples of types of objects available in R. In Section 1.4 we go through the most common objects in R.

1.4 Types and classes of objects

Objects can have classes, which allow functions to interact with them. Objects can be of several classes. We already used the class `numeric`, which is used for general numbers, but there are also additional very commonly used classes

- `integer`, for integer numbers
- `character`, just for character strings
- `factor`, used to represent levels of a categorical variable
- `logical`, the values TRUE and FALSE

While many others exist, these are the more commonly used. Outputs of some analyses have special classes, as an example, the output of a call of function `lm()` is an object of class `lm`, i.e., a linear model. Typically, functions behave differently according to the class of an object. As an example, note how `summary()` treats differently an object of class `factor` or one of class `numeric`, producing a table of counts per level for a factor but a 6 number summary for numeric values.

```
obj1 <- factor(c(rep("a",12), rep("b",4), rep("c",2)))
summary(obj1)
```

```
##  a  b  c
## 12  4  2
```

```
obj2 <- c(2,5,-0.2,89,12,-3,-5.4)
summary(obj2)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    -5.4    -1.6     2.0    14.2     8.5    89.0
```

We can check the class of an object using function `class`, as in the following examples

```
class(obj1)
```

```
## [1] "factor"
```

```
class(obj2)
```

```
## [1] "numeric"
```

```
class(TRUE)
```

```
## [1] "logical"
```

It is sometimes useful to coerce objects into different classes, but care should be used when doing so. Some examples are presented below. Can you describe in your own words what R did below?

```
as.integer(c(3,-0.3,0.4,0.6,0.9,13.2,12))
```

```
## [1]  3  0  0  0  0 13 12
```

```
as.numeric(c(TRUE,FALSE,TRUE))
```

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

```
as.numeric(obj1)
```

```
##  [1] 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 3 3
```

A common way to organize multiple vectors together is in the form of a matrix. Here we create such an object

```
mat1 <- matrix(1:12, nrow=3, ncol=4)
mat1
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

Note that by default R fills the first column (with 1,2,3) then the second column (4,5,6) etc. If you want it to fill the first row, then the second, you can use the optional argument `byrow=TRUE`, like this:

```
matrix(1:12, nrow=3, ncol=4, byrow=TRUE)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12
```

R also allows data structures with more than 2 dimensions – we don't cover those here, but look up the help on `array` if you're interested. A matrix is just a two dimensional array.

Arrays are useful objects, but can be complex to visualize due to their potential high dimensionality. Another common type of object is a `data.frame`. This is essentially a matrix but for which each column can be of a different type. These are what we would typically associate with an excel spreadsheet or a table in a database. Typically columns correspond to variables observed in a number of subjects, each subject recorded in its own row. A simple example with 3 variables and 5 subjects follows:

```
mysex <- c("male","female","female","male","male")
myage <- c(34,23,56,45,12)
myhei <- c(185,178,167,165,148)
df1 <- data.frame(ID=1:5, sex=mysex, age=myage, height=myhei)
df1
```

```
##   ID    sex age height
## 1  1   male  34    185
## 2  2 female  23    178
## 3  3 female  56    167
## 4  4   male  45    165
## 5  5   male  12    148
```

Typically, `data.frames` are used to store the data we subsequently analyse. Usually the data are not manually imputed as above, but read into R from other software, using R functions addressed in a later section.

A data frame is just a special type of `list`. A `list` can contain objects of different types and dimensions. An example is here

```
list1 <- list(Note="whatever I want here", X2=4, age=1:4)
list1
```

```
## $Note
## [1] "whatever I want here"
##
## $X2
## [1] 4
##
## $age
## [1] 1 2 3 4
```

Lists are typically used to store outputs of computations which require different kinds of objects to be recorded. Note the use of $ to access the sub-components of a list or a data.frame.

```
list1$X2+10
```

```
## [1] 14
```

A final type of object which we already used are functions. While there are thousands of available functions inside R, later we will learn how to create our own functions.

1.5 Subsetting data

One useful feature of R relates to how we can index subsets of data. The indexing information is included within square brackets: `[]`. As an example, we can select the third element of a vector

```
x<-c(1,3.5,7,8,-7,0.43,-1)
x[3]
```

```
## [1] 7
```

but we can also select all *except* the second and third elements of the same vector

```
x[-c(2,3)]
```

```
## [1]  1.00  8.00 -7.00  0.43 -1.00
```

We can also select only the objects which follow a given condition, say only those that are positive

```
x[x>0]
```

```
## [1] 1.00 3.50 7.00 8.00 0.43
```

or those between (-1,1)

```
x[(x>-1) & (x<1)]
```

```
## [1] 0.43
```

Note the subtle difference between the previous and next statements

```
x[(x>=-1) & (x<=1)]
```

```
## [1]  1.00  0.43 -1.00
```

which reminds us we should be careful when setting these logical conditions, especially when working with integer boundaries which might be on the limits of those conditions. Note indexing can be done using additional information. As an example, we select here the elements in `x` such that the corresponding elements in `y` are positive:

```
#rnorm(k) produces k Gaussian random deviates
x <- rnorm(10)
y <- rnorm(10)
x2 <- x[y>0]
```

When working on a matrix the indexing is done by row and column, therefore for selecting the value that is in the third row and second column of a matrix we use

```
mat1[3,2]
```

```
## [1] 6
```

but we can also select all the elements in the second row

```
mat1[2,]
```

```
## [1]  2  5  8 11
```

or the fourth column

```
mat1[,4]
```

```
## [1] 10 11 12
```

1.6 Mathematical functions and simple data calculations

Within R there are a number of mathematical operators but also mathematical and statistical functions. As any other functions, many of these have required parameters and optional parameters. It would take a very long time to describe even the most basic functions. Therefore, we prefer to let you try hands on explore a number of these.

Task 1: Take your time to explore the functions below:

`sum(x)`	`sqrt(x)`	`log(x)`	`log(x,n)`	`exp(x)`	`choose(n,x)`
`factorial(x)`	`floor(x)`	`ceiling(x)`	`round(x,digits)`	`abs(x)`	`cos(x)`
`sin(x)`	`tan(x)`	`acos(x)`	`acosh(x)`	`max(x)`	`min(x)`
`mean(x)`	`median(x)`	`range(x)`	`var(x)`	`cor(x,y)`	`quantile(x)`

(Tip: do not forget that you can get a full description what each function can be used for, what arguments it takes, and what kind of output it produces, using "?". Further, the help of most functions

includes examples of their use, which proves invaluable to understand their usage.)

1.7 Importing and exporting data

Rather than importing data into R manually, typically the data we work with are imported from some external source. Typically this might be some simple file format, like a txt or a csv file, but while not covered here, direct import from say Excel files or Access data bases is possible. Such more specialized inputs often require additional packages.

RStudio includes a useful dedicated shortcut "Import dataset", by default available through the top right window of RStudio's interface. Note this shortcut essentially just calls the appropriate functions required for each import. Here we present a couple of examples just for practising.

First, we load up a data frame which exists in R[4] and contains an example data set, with variables measured in 150 flowers of 3 varieties. This is in object `iris`, and we use the function `data()` to load it so that we have access to it.

```
data(iris)
```

we can take a look at what this data set contains

```
# example of head use: see the first 4 rows in iris
head(iris, n=4)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
```

```
# example of str use
str(iris)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

```
# example of summary use
summary(iris)
```

```
##   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
##  Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
##  1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
##  Median :5.800   Median :3.000   Median :4.350   Median :1.300
##  Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
##  3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
##  Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
```

[4] R includes a large variety of example data sets which are useful to illustrate the use of code.

```
##        Species
##  setosa    :50
##  versicolor:50
##  virginica :50
##
##
##
```

Now we create a new data frame which we then modify to include a new variable

```
mydata <- iris
mydata$total <- mydata$Sepal.Length + mydata$Sepal.Width + mydata$Petal.Length + mydata$Petal.Width
```

Now, we are going to export this data set as a txt, named `mydatafile.txt`

```
write.table(mydata, file="mydatafile.txt", row.names=FALSE)
```

Note the use of the optional argument `row.names=FALSE`, otherwise some arbitrary row names would be added to the file. If you look in the folder you are working in, you should now have a new file there. Open it and check that it looks as you would expect. Next, we are going to import it back into R, into an object named `indat`.

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

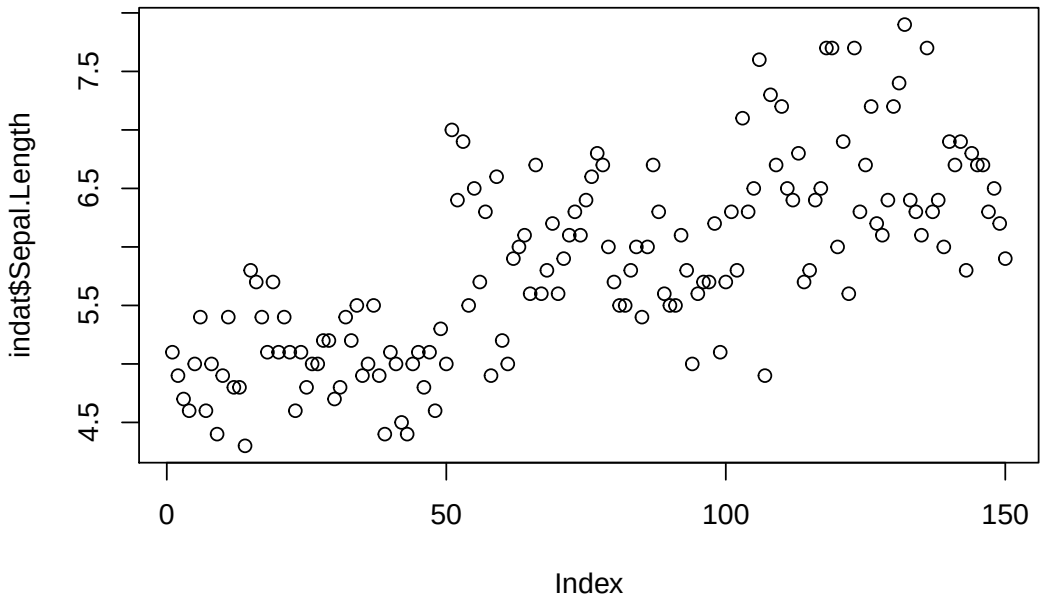
So now we have our data back in R.

Task 2: Import the file `dados1.csv` into R, giving it the name `newfile`. Tips: Explore the possible options including 1. `Import Dataset` shortcut in the `Environment tab`, 2. the optional argument `sep=","` in function `read.table` or 3. consider using function `read.csv()`.

1.8 Graphics

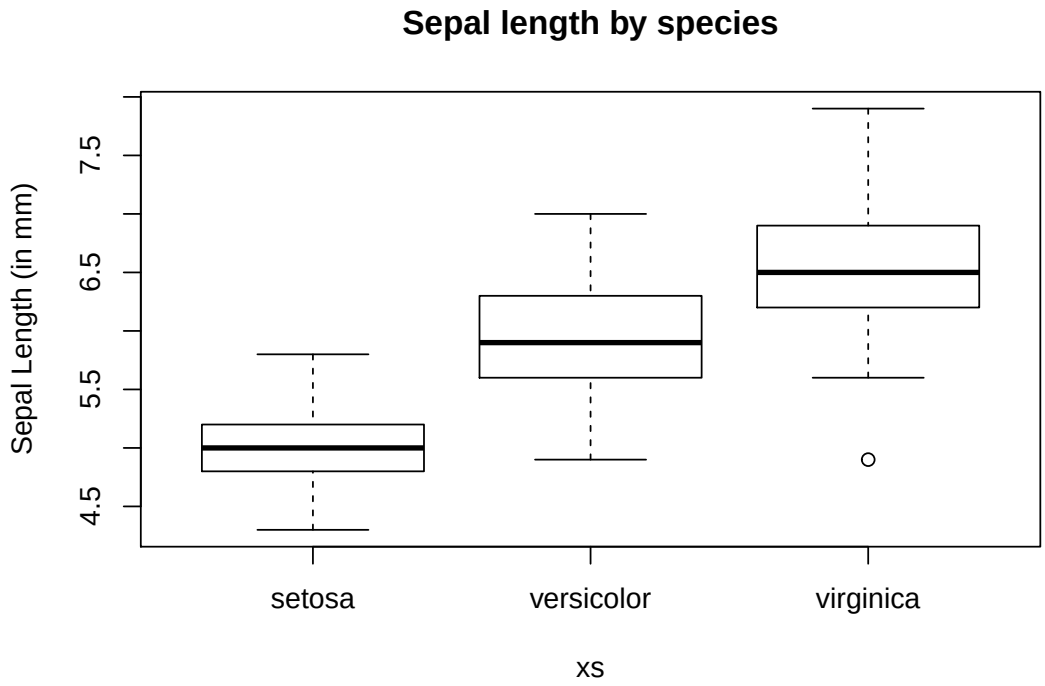
One of the most amazing R capabilities are its graphics customization properties. One can create pretty much any graphic output desirable. The plot function is, as we have seen before for function `summary()`, a function that attempts to do something smart depending on the type of arguments used. Using the data set iris previously considered, plot examples are implemented below, with some optional arguments being used to show some of the possibilities to customize plots.

```
#default use
plot(indat$Sepal.Length)
```



We now add some labels to a new plot of sepal length as a function of species (note the use of ~ to mean *as a function of*; this is also used below when specifying regression models, where the object on the left of ~ will be the response variable and the objects on the right explanatory variables)

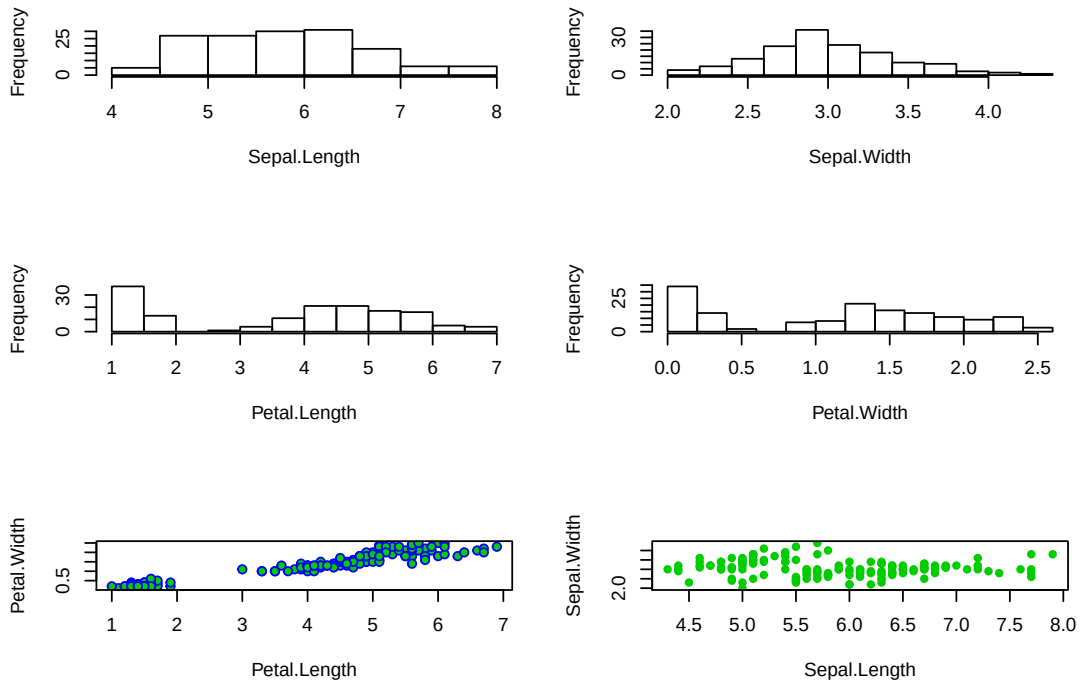
```
ys <- indat$Sepal.Length
xs <- indat$Species
#note use of ~ to represent "as a function of"
plot(ys~xs, ylab="Sepal Length (in mm)", main="Sepal length by species")
```



We can also set the graphic window to hold multiple plots. This is obtained via argument `mfrow`, one of the arguments in function `par`.[5] An example follows, in which we leverage on the use of function `with` to avoid having to constantly use `indat$` to tell R where the data can be found.

```
#define two rows and 2 columns of plots
par(mfrow=c(3,2))
with(indat, hist(Sepal.Length, main=""))
with(indat, hist(Sepal.Width, main=""))
with(indat, hist(Petal.Length, main=""))
with(indat, hist(Petal.Width, main=""))
with(indat, plot(Petal.Length, Petal.Width, pch=21, col=12, bg=3))
with(indat, plot(Sepal.Length, Sepal.Width, pch=16, col=3))
```

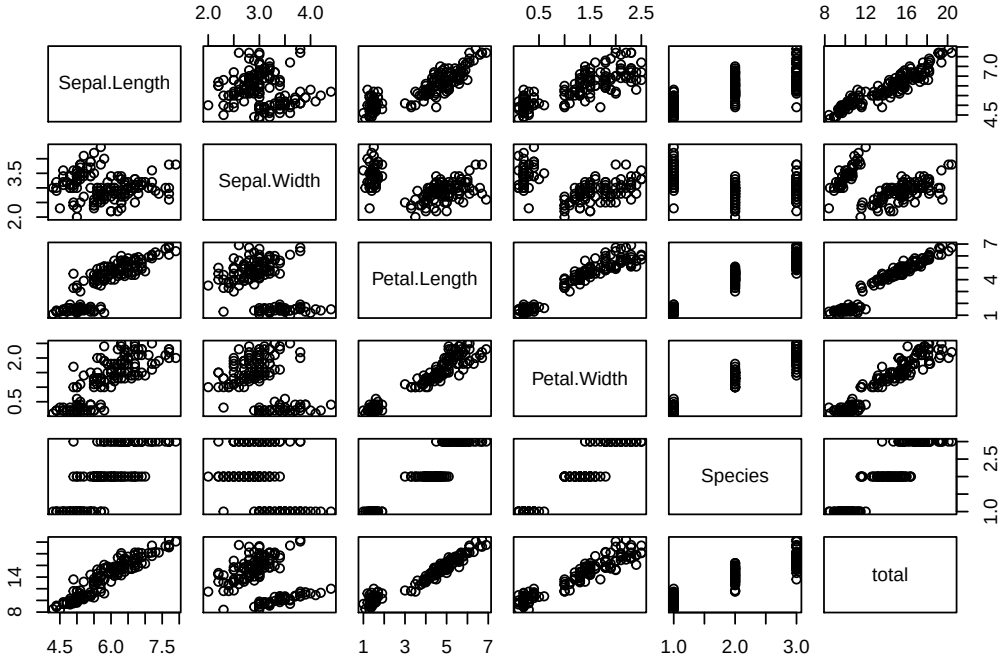
[5] Note this function controls a much larger number of graphical parameters. You can take a look at its help file to get a feel for how many and what kind of control it allows you.



We used argument `mfrow`, but looking at the help for function ‘par gives you an insight to the level of customization one can reach with respect to these graphical parameters, via dozens of different arguments.

We can look at the correlation structure between all variables using function `pairs()`.

```
# define two rows and 2 columns of plots
par(mfrow=c(1,1))
pairs(indat)
```



Task 3: Using data `cars`, create a plot that represents the stopping distances as a function of the speed of cars. Use the `points` function to add a special symbol to points corresponding to cars with speed lower than 15 mph, but distance larger than 70m. Check out the function `text` to add text annotations to plots. Customize axis labels.

1.9 Extending basic capabilities via packages

While R base installation includes enough functions that getting acquainted with them could take several years, many more are available via the installation of additional packages available online. A package is just a set of functions and data sets (and the corresponding documentation plus some additional required files) which usually have some specific goal. As examples, in our workshop we will be using packages `secr` and `mgcv`, which allow the implementation of spatially explicit capture recapture (SECR) models and generalized additive models (GAM), respectively.

Note packages cover a very wide range of applications, and chances are that at least a package, often more than one, already exists to implement most kinds of statistical or data processing tasks we might imagine.

Installing a new package in R requires a call to function `install.packages()`. A RStudio shortcut is simply to follow the `Tools|Install packages...` shortcut.

After a package is installed it needs to be loaded to be available. In R this is done calling function `library()` with the package name as an argument. In RStudio this becomes simpler by checking the

boxes under the RStudio tab packages (by default this tab is available on the bottom right window, along with the Files, Plots, Help and Viewer tabs).

We use `secr` as an example. Notice, to begin with `secr` is not available

```
?secr
```

```
## No documentation for 'secr' in specified packages and libraries:
## you could try '??secr'
```

Next, we install the package.

```
install.packages("secr")
```

Then, we load the package

```
library("secr")
```

```
## This is secr 3.0.1. For overview type ?secr
```

and finally we check that the functions in it are now loaded

```
?secr
```

We would now be ready to analyse results from a SECR survey.

Task 4: Run the example code available in the help page from package `secr`. Try to understand what is happening: we simulate some SECR data and we then estimate density based on simulated capture histories. In particular, look at the simulated density and the estimated density. This is just a taster for the course to follow. . . !

1.10 Linear regression

One of the most common type of data analysis is a regression model. Despite common and conceptually simple, it is a very powerful way to understand which (and how) of a number of candidate variables, sometimes referred to covariates, independent or explanatory variables, might influence a dependent variable, also often referred as the response. There are many flavours of regression models, from a simple linear regression to complicated generalized additive mixed models. We do not wish to present these in any detail, but to introduce you to some functions that implement these models and the syntax that R uses to describe them.

Let's start with the basics. You have used the `cars` data set above. We use it here again to try to explain the distance a car takes to stop as a function of its speed. We start with a linear model using function `lm()`

```
data(cars)
mylm1 <- lm(dist~speed, data=cars)
```

We have stored the result of fitting the model in object `mylm1`. The function `summary()` can be used to print a summary of the fit

```
summary(mylm1)
```

```
##
## Call:
```

```
## lm(formula = dist ~ speed, data = cars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -29.069  -9.525  -2.272   9.215  43.201
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -17.5791     6.7584  -2.601   0.0123 *
## speed         3.9324     0.4155   9.464 1.49e-12 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 15.38 on 48 degrees of freedom
## Multiple R-squared:  0.6511, Adjusted R-squared:  0.6438
## F-statistic: 89.57 on 1 and 48 DF,  p-value: 1.49e-12
```

Do not get frightened about all the output. The coefficient associated with speed tells us what intuition alone would anticipate, the higher the speed, the larger the distance a car takes to stop. The easier way to see the relationship is by adding a line to the plot (note this is a similar plot to what you should have created in task 3 above!). The predicted relationship is shown in Figure 1.1.

```
xl <- "Speed (mph)"
yl <- "Distance (m)"
plot(cars$speed, cars$dist, xlab=xl, ylab=yl, ylim=c(0,120) ,xlim=c(0,30))
abline(mylm1)
```

Note how function `abline()` is used with a linear model as its first argument and it uses the parameters in said object to add a line to the plot. The optional arguments `v` and `h` are often very useful to draw vertical and horizontal lines in plots.

Task 5: Use abline to draw dashed lines (tip, use optional argument `lty=2`) representing the estimated distance that a car moving at 16 mph would take to stop.

Note that the line added to the plot represents the distance a car would take to stop given its speed. Oddly enough, it seems like a car going at 3 mph might take a negative time to stop, which is just plain nonsense. Why? Because we used a model which does not respect the features of the data. A stopping distance can not be negative. However, implicit in the linear model we used, distance is a Gaussian (=normal) random variable. We can avoid this by using a generalized linear model (GLM). Now the response can have a range of distributions. An example of such distribution that takes only positive values is the gamma distribution. We implement a gamma GLM next

```
#fit the glm
myglm1 <- glm(dist~speed, data=cars, family=Gamma(link=log))
#predict using the glm for speeds between 1 and 30
predmyglm1 <- predict.glm(myglm1, newdata=data.frame(speed=1:30),
                          type="response")
```

Our model now assumes the response has a gamma distribution, and the link function is the logarithm. The link function allows you to change how the mean value is related to the covariates. This becomes rather technical rather fast. Details about glms are naturally beyond the scope of this tutorial. References like Faraway (2006) or Zuur et al. (2009) will provide further details in an applied context.

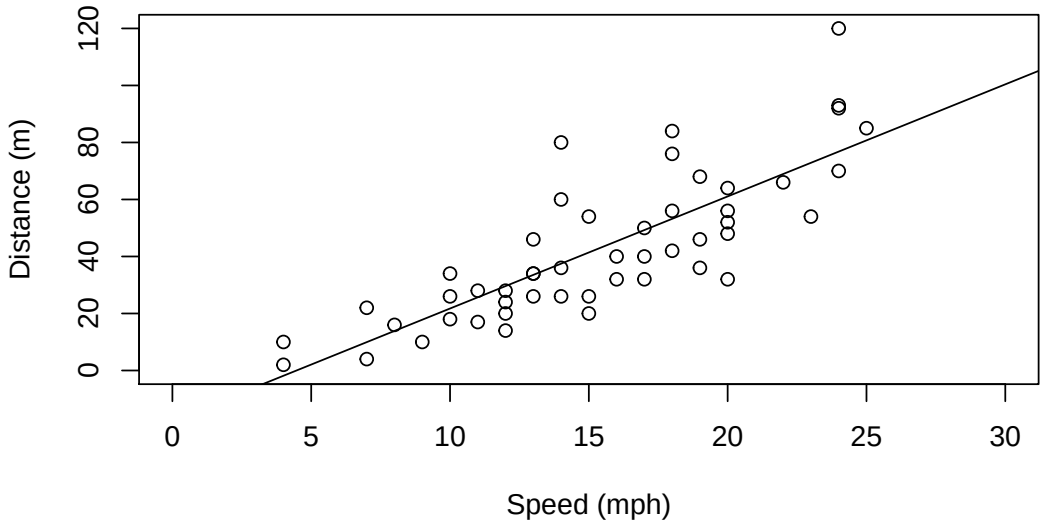


Figure 1.1: The data and the linear fit added to it.

The predicted relationship is shown in Figure 1.2.

```
#create a plot
plot(cars$speed,cars$dist, xlab="Speed (mph)", ylab="Distance (m)",
     ylim=c(0,120), xlim=c(0,30))
#add the linear fit
abline(mylm1)
#and now add the glm predictions
lines(1:30, predmyglm1, col="blue", lwd=3, lty=3)
```

However, this glm still requires that the response is linear at some scale (in this case, on the scale of the link function). Sometimes, non-linear effects are present. These can be fitted using generalized additive models. A good introduction to GAMs is provided by Wood (2006) and Zuur et al. (2009).

So finally we fit a gam model to the same data set. For that we require library `mgcv`. The outcome is shown in Figure 1.3. Here the fit is not very different from the glm fit, but under many circumstances a gam might be required over a glm. We will see such an example in the next few days, when we model the detectability of beaked whale clicks as a function of distance and angle (with respect to hydrophones).

```
#load the mgcv library
library(mgcv)
```

```
## Loading required package: nlme
```

```
## This is mgcv 1.8-17. For overview type 'help("mgcv-package")'.
```

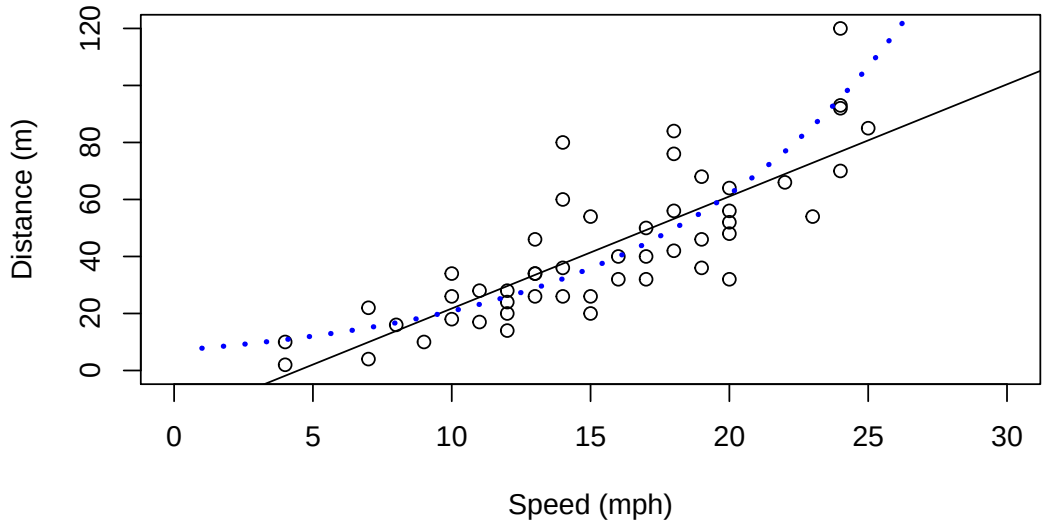


Figure 1.2: The data and the linear and Gamma glm fits added to it.

```
#fit the gam
mygam1 <- gam(dist~s(speed), data=cars, family=Gamma(link=log))
#predict using the glm for speeds between 1 and 30
predmygam1 <- predict(mygam1, newdata=data.frame(speed=1:30), type="response")
```

```
#create a plot
plot(cars$speed, cars$dist, xlab="Speed (mph)", ylab="Distance (m)",
     ylim=c(0,120), xlim=c(0,30))
#add the linear fit
abline(mylm1)
#and now add the glm predictions
lines(1:30, predmyglm1, col="blue", lwd=3, lty=3)
lines(1:30, predmygam1, col="green", lwd=3, lty=2)
```

1.11 Some advanced capabilities of R

1.11.1 Simulation and random number generation

Another powerful use of R is for simulation. To this end, R has the ability to simulate random deviates from a large number of distributions. Perhaps the more useful and commonly used are the uniform and the Gaussian distributions. We now create 50 random deviates from each of these, as well as

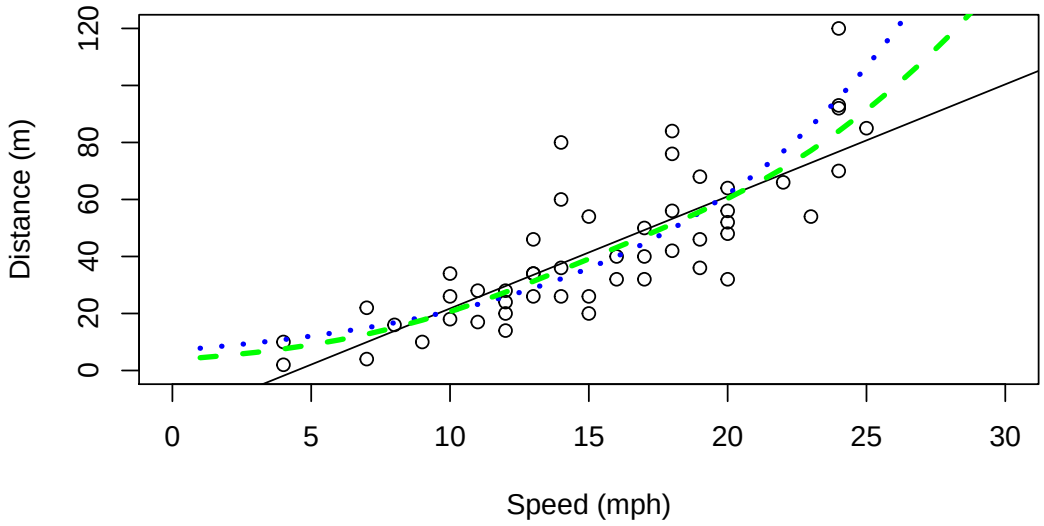


Figure 1.3: The data and the linear and Gamma glm and gam fits added to it.

some Poisson deviates, for illustration

```
#define two rows and 2 columns of plots
rdnorm <- rnorm(50,mean=20,sd=3)
rdunif <- runif(50,min=3,max=6)
rdpois <- rpois(50,lambda=6)
```

R can create random numbers from many different distributions (see help(Distributions) for a list) – the relevant functions generally start with r and then an abbreviated distribution name (rbinom, rexp, rgeom, etc). Additionally, R also includes the ability to obtain the density function, distribution function and quantile function via the `d`+name, `p`+name and `q`+name functions. As an example, the Gaussian function usage of these functions is presented below

```
dnorm(0,mean=0,sd=1)
```

```
## [1] 0.3989423
```

```
pnorm(0,mean=0,sd=1)
```

```
## [1] 0.5
```

```
qnorm(0.975,mean=0,sd=1)
```

```
## [1] 1.959964
```

Task 6: Using what you have learnt here, create two histograms, one of 50, another of 5000, random deviates from a Gaussian distribution, using the optional argument `freq=FALSE` (leading to an estimate

of the density function). Then add a line to the plot that represents the true underlying density (tip, you can use function `dnorm()`), and comment on the results.

1.11.2 Writing your own functions

While the above functions, and the many more available, make R a very useful tool, there are sometimes problems which require a special tool. For these, we can create our own functions. Note this is an advanced topic.

The way of doing that follows a specific syntax

```
> name <- function(arg1,arg2,...) {what the function does goes here}
```

As an example, we create a function that returns the sum of its arguments:

```
myfun <- function(i,j){
  myres <- i + j
  return(myres)
}
```

Task 7: create a function called `mystats()` which returns the mean, variance, maximum and minimum of the first argument (a vector). Then, update your function such that it can also return the mean excluding the negative numbers.

1.12 Wrap up

A full introduction to R course could take an entire week. A full course in regression modelling with R could take an entire semester. A full course of data analysis in R could take a life time.

Our objective with this tutorial was simply to introduce you to R such that when we use R in the next few days, the commands do not look too esoteric. Nonetheless, this material as well as the references provided should constitute a good basis to learn R further if you so desire. Beginners find the learning curve is often steep, but once mastered, R simplifies enormously the task of statistical data analysis.

```
#cleaning the workspace
rm(list = ls())
```

Chapter 2

analyse simulated data set

2.1 Laura to prepare

Chapter 3

Problem datasets

3.1 Len to prepare

Chapter 4

Simulation of distance sampling data

4.1 Laura to prepare

4.2 Bit of exposure to survey design engine

4.3 Trails/no trails exercise from previous years

4.4 Perhaps some exposure to DSsim vignette

Chapter 5

Preparing data for analysis

5.1 Aims

By the end of this practical, you should feel comfortable:

- Loading data from a geodatabase file into R
- Removing and renaming columns in a `data.frame`
- Saving data to an `RData` file

Note we can (and should) re-run this file when we update the `Analysis.gdb` file to ensure that the data R uses has all of the covariates we want to use in our analysis.

5.2 Preamble

Load some useful packages:

```
library(rgdal)
library(knitr)
```

5.3 Load and arrange data

To fit our spatial models we require three objects:

1. The detection function we fitted previously.
2. The segment data (sometimes called effort data). This tells us how much effort was expended per segment (in this case how far the boat went) and includes the covariates that we want to use to fit our model.
3. The observation table. This links the observations in the detection function object to the segments.

In R we can use the `rgdal` package to access the geodatabase files generated by ArcGIS (R can also access shapefiles and rasters).

It can be useful in general to see which "layers" are available in the geodatabase, for that we can use the `ogrListLayers()` function:

```
ogrListLayers("Analysis.gdb")
```

```
##  [1] "EN_Trackline1"     "EN_Trackline2"     "GU_Trackline2"
##  [4] "GU_Sightings"      "EN_Sightings"      "GU_Trackline"
##  [7] "Tracklines"        "Tracklines2"       "Segments"
## [10] "Sightings"         "Segment_Centroids" "Study_Area"
## [13] "US_Atlantic_EEZ"
## attr(,"driver")
## [1] "OpenFileGDB"
## attr(,"nlayers")
## [1] 13
```

5.3.1 Segment data

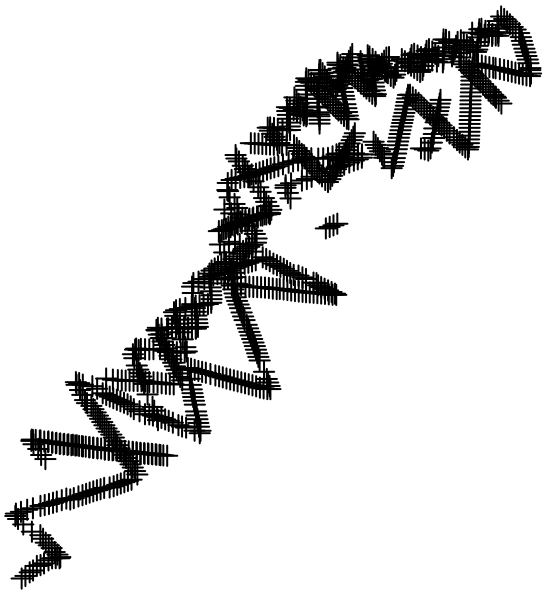
For our analysis the segment data is located in in the "Segment_Centroids" table in the geodatabase. We can import that into R using the `readOGR()` function:

```
segs <- readOGR("Analysis.gdb", layer="Segment_Centroids")
```

```
## OGR data source with driver: OpenFileGDB
## Source: "Analysis.gdb", layer: "Segment_Centroids"
## with 949 features
## It has 10 fields
```

To verify we have the right data we can plot it. This will give the locations of each segment:

```
plot(segs)
```



A further check would be to use `head()` to check that the structure of the data is correct. In particular it's worth checking that the column names are correct and that the number of rows in the data set are correct (`dim()` will give the number of rows and columns).

It can also be useful to check that the columns are the correct data types. Calling `str(segs@data)` (or any object loaded using `readOGR` appended with `@data`) will reveal the data types of each column. In this case we can see that the `CenterTime` column has been interpreted as a `factor` variable rather than as a date/time. We're not going to use it in our analysis, so we don't need to worry for now but `str()` can reveal potential problems with loaded data.

For a deeper look at the values in the data, `summary()` will give summary statistics for each of the covariates as well as the projection and range of location values (lat/long or in our case `x` and `y`). We can compare these with values in ArcGIS.

We can turn the object into a `data.frame` (so R can better understand it) and then check that it looks like it's in the right format using `head()`:

```
segs <- as.data.frame(segs)
head(segs)
```

```
##            CenterTime SegmentID   Length  POINT_X  POINT_Y     Depth
## 1 2004/06/24 07:27:04         1 10288.91 214544.0 689074.3  118.5027
## 2 2004/06/24 08:08:04         2 10288.91 222654.3 682781.0  119.4853
## 3 2004/06/24 09:03:18         3 10288.91 230279.9 675473.3  177.2779
## 4 2004/06/24 09:51:27         4 10288.91 239328.9 666646.3  527.9562
## 5 2004/06/24 10:25:39         5 10288.91 246686.5 659459.2  602.6378
## 6 2004/06/24 11:00:22         6 10288.91 254307.0 652547.2 1094.4402
##    DistToCAS      SST          EKE      NPP coords.x1 coords.x2
## 1 14468.1533 15.54390 0.0014442616 1908.129  214544.0  689074.3
## 2 10262.9648 15.88358 0.0014198086 1889.540  222654.3  682781.0
## 3  6900.9829 16.21920 0.0011704842 1842.057  230279.9  675473.3
## 4  1055.4124 16.45468 0.0004101589 1823.942  239328.9  666646.3
## 5  1112.6293 16.62554 0.0002553244 1721.949  246686.5  659459.2
## 6   707.5795 16.83725 0.0006556266 1400.281  254307.0  652547.2
```

As with the distance data, we need to give the columns of the data particular names for them to work with `dsm`:

```
segs$x <- segs$POINT_X
segs$y <- segs$POINT_Y
segs$Effort <- segs$Length
segs$Sample.Label <- segs$SegmentID
```

5.3.2 Observation data

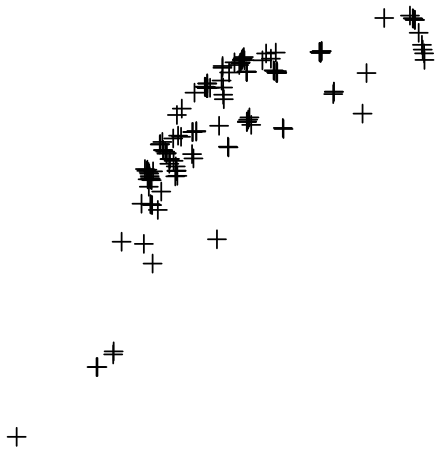
The observation data is exactly what we used to fit out detection function in the previous exercise (though this is not necessarily always true).

```
obs <- readOGR("Analysis.gdb", layer="Sightings")
```

```
## OGR data source with driver: OpenFileGDB
## Source: "Analysis.gdb", layer: "Sightings"
## with 137 features
## It has 7 fields
```

Again we can use a plot to see whether the data looks okay. This time we only have the locations of the observations:

```
plot(obs)
```



Again, converting the object to be a `data.frame` and checking it's format using `head()`:

```
obs <- as.data.frame(obs)
head(obs)
```

```
##    Survey GroupSize SeaState  Distance        SightingTime SegmentID
## 1 en04395         2      3.0  246.0173 2004/06/28 10:22:21        48
## 2 en04395         2      2.5 1632.3934 2004/06/28 13:18:14        50
## 3 en04395         1      3.0 2368.9941 2004/06/28 14:13:34        51
## 4 en04395         1      3.5  244.6977 2004/06/28 15:06:01        52
## 5 en04395         1      4.0 2081.3468 2004/06/29 10:48:31        56
## 6 en04395         1      2.4 1149.2632 2004/06/29 14:35:33        59
##   SightingID coords.x1 coords.x2
## 1          1   -65.636    39.576
## 2          2   -65.648    39.746
## 3          3   -65.692    39.843
## 4          4   -65.717    39.967
## 5          5   -65.820    40.279
## 6          6   -65.938    40.612
```

Finally, we need to rename some of the columns:

```
obs$distance <- obs$Distance
obs$object <- obs$SightingID
obs$Sample.Label <- obs$SegmentID
obs$size <- obs$GroupSize
```

5.4 Save the data

We can now save the `data.frame`s that we've created into an `RData` file so we can use them later.

```
save(segs, obs, file="sperm-data.RData")
```

Chapter 6

Detection function fitting

6.1 Aims

By the end of this practical you should feel confident doing the following:

- Loading data from ArcGIS `.gdb` files
- Working on a `data.frame` in R to get it into the correct format for `Distance`
- Fitting a detection function using `ds()`
- Checking detection functions
- Making at goodness of fit plots
- Selecting models using AIC
- Estimating abundance (using R and maths!)

6.2 Preamble

First need to load the requisite R libraries

```
library(rgdal)
library(ggplot2)
library(Distance)
```

```
## Loading required package: mrds
```

```
## This is mrds 2.1.18
## Built: R 3.4.0; ; 2017-06-12 11:05:22 UTC; windows
```

```
##
## Attaching package: 'Distance'
```

```
## The following object is masked from 'package:mrds':
##
##     create.bins
```

```
library(knitr)
library(kableExtra)
```

6.3 Load the data

The observations are located in a "geodatabase" we created in Arc. We want to pull out the "Sightings" table (called a "layer") and make it into a `data.frame` (so it's easier for R to manipulate).

```
distdata <- readOGR("Analysis.gdb", layer="Sightings")
```

```
## OGR data source with driver: OpenFileGDB
## Source: "Analysis.gdb", layer: "Sightings"
## with 137 features
## It has 7 fields
```

```
distdata <- as.data.frame(distdata)
```

We can check it has the correct format using `head`:

```
head(distdata)
```

```
##    Survey GroupSize SeaState  Distance        SightingTime SegmentID
## 1 en04395         2      3.0  246.0173 2004/06/28 10:22:21        48
## 2 en04395         2      2.5 1632.3934 2004/06/28 13:18:14        50
## 3 en04395         1      3.0 2368.9941 2004/06/28 14:13:34        51
## 4 en04395         1      3.5  244.6977 2004/06/28 15:06:01        52
## 5 en04395         1      4.0 2081.3468 2004/06/29 10:48:31        56
## 6 en04395         1      2.4 1149.2632 2004/06/29 14:35:33        59
##   SightingID coords.x1 coords.x2
## 1          1   -65.636    39.576
## 2          2   -65.648    39.746
## 3          3   -65.692    39.843
## 4          4   -65.717    39.967
## 5          5   -65.820    40.279
## 6          6   -65.938    40.612
```

The `Distance` package expects certain column names to be used. Renaming is much easier to do in R than ArcGIS, so we do it here.

```
distdata$distance <- distdata$Distance
distdata$object <- distdata$SightingID
distdata$size <- distdata$GroupSize
```

Let's see what we did:

```
head(distdata)
```

```
##    Survey GroupSize SeaState  Distance        SightingTime SegmentID
## 1 en04395         2      3.0  246.0173 2004/06/28 10:22:21        48
## 2 en04395         2      2.5 1632.3934 2004/06/28 13:18:14        50
## 3 en04395         1      3.0 2368.9941 2004/06/28 14:13:34        51
```

```
## 4 en04395         1      3.5  244.6977 2004/06/28 15:06:01        52
## 5 en04395         1      4.0 2081.3468 2004/06/29 10:48:31        56
## 6 en04395         1      2.4 1149.2632 2004/06/29 14:35:33        59
##   SightingID coords.x1 coords.x2  distance object size
## 1          1   -65.636    39.576  246.0173      1    2
## 2          2   -65.648    39.746 1632.3934      2    2
## 3          3   -65.692    39.843 2368.9941      3    1
## 4          4   -65.717    39.967  244.6977      4    1
## 5          5   -65.820    40.279 2081.3468      5    1
## 6          6   -65.938    40.612 1149.2632      6    1
```

We now have four "extra" columns.

6.4 Exploratory analysis

Before setting off fitting detection functions, let's look at the relationship of various variables in the data.

Don't worry too much about understanding the code that generates these plots at the moment.

6.4.1 Distances

Obviously, the most important covariate in a distance sampling analysis is distance itself. We can plot a histogram of the distances to check that (1) we imported the data correctly and (2) it conforms to the usual shape for line transect data.

```
hist(distdata$distance, xlab="Distance (m)", main="Distance to sperm whale observations")
```

6.4.2 Size and distance

We might expect that there will be a relationship between the distance at which we see animals and the size of the groups observed (larger groups are easier to see at larger distances), so let's plot that to help us visualise the relationship.

```
# plot of size versus distance and sea state vs distance, linear model and LOESS smoother overlay

# put the data into a simple format, only selecting what we need
distplot <- distdata[,c("distance","size","SeaState")]
names(distplot) <- c("Distance", "Size", "Beaufort")
library(reshape2)
# "melt" the data to have only three columns (try head(distplot))
distplot <- melt(distplot, id.vars="Distance", value.name="covariate")

# make the plot
p <- ggplot(distplot, aes(x=covariate, y=Distance)) +
      geom_point() +
      facet_wrap(~variable, scale="free") +
```

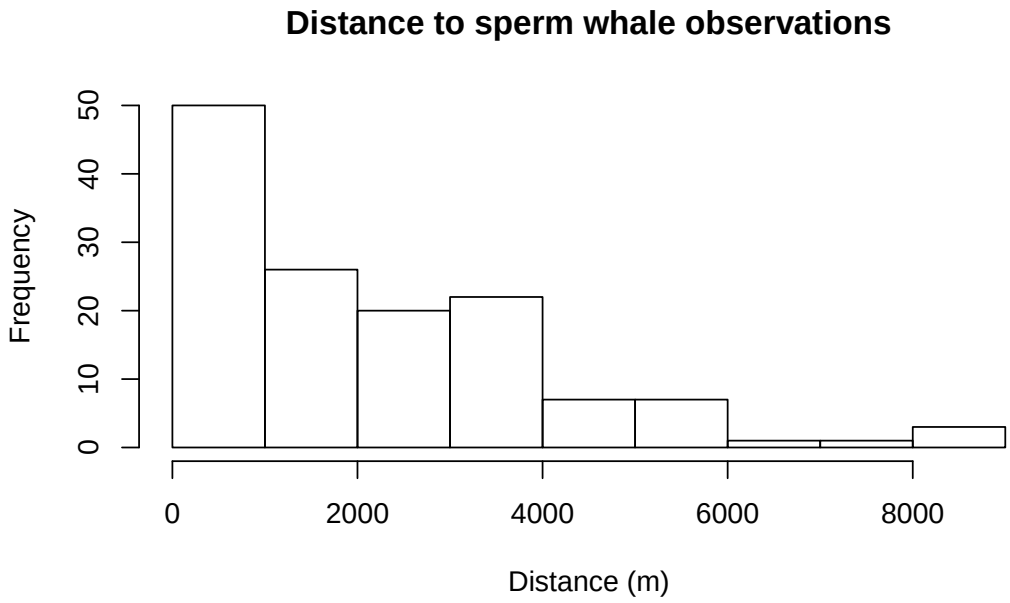


Figure 6.1: Distribution of observed perpendicular detection distances.

```
      geom_smooth(method="loess", se=FALSE) +
      geom_smooth(method="lm", se=FALSE) +
      labs(x="Covariate value", y="Distance (m)")
print(p)
```

6.4.3 Distance and sea state

We might also expect that increasing sea state would result in a drop in observations. We can plot histograms of distance for each sea state level (making the sea state take only values 0,1,2,4,5 for this).

```
distdata$SeaStateCut <- cut(distdata$SeaState,seq(0,5,by=1), include.lowest=TRUE)
p <- ggplot(distdata) +
      geom_histogram(aes(distance)) +
      facet_wrap(~SeaStateCut) +
      labs(x="Distance (m)", y="Count")
print(p)
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

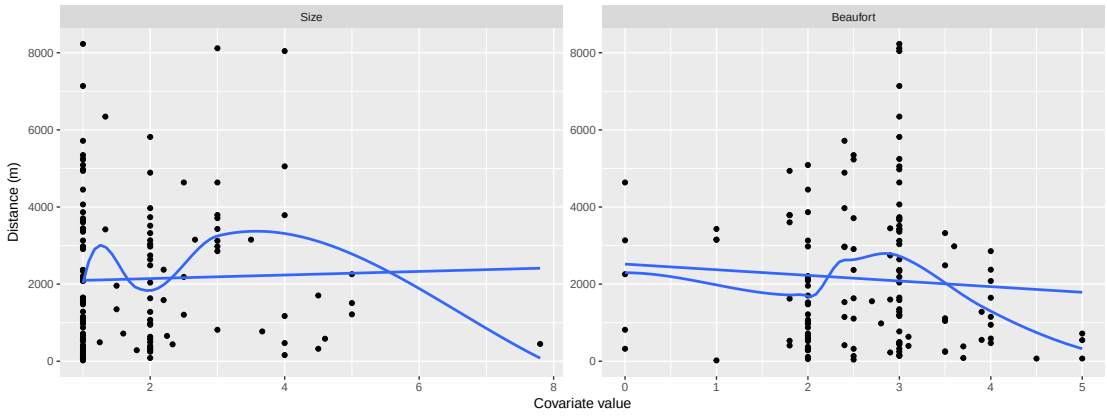


Figure 6.2: Effect of group size upon detection distances.

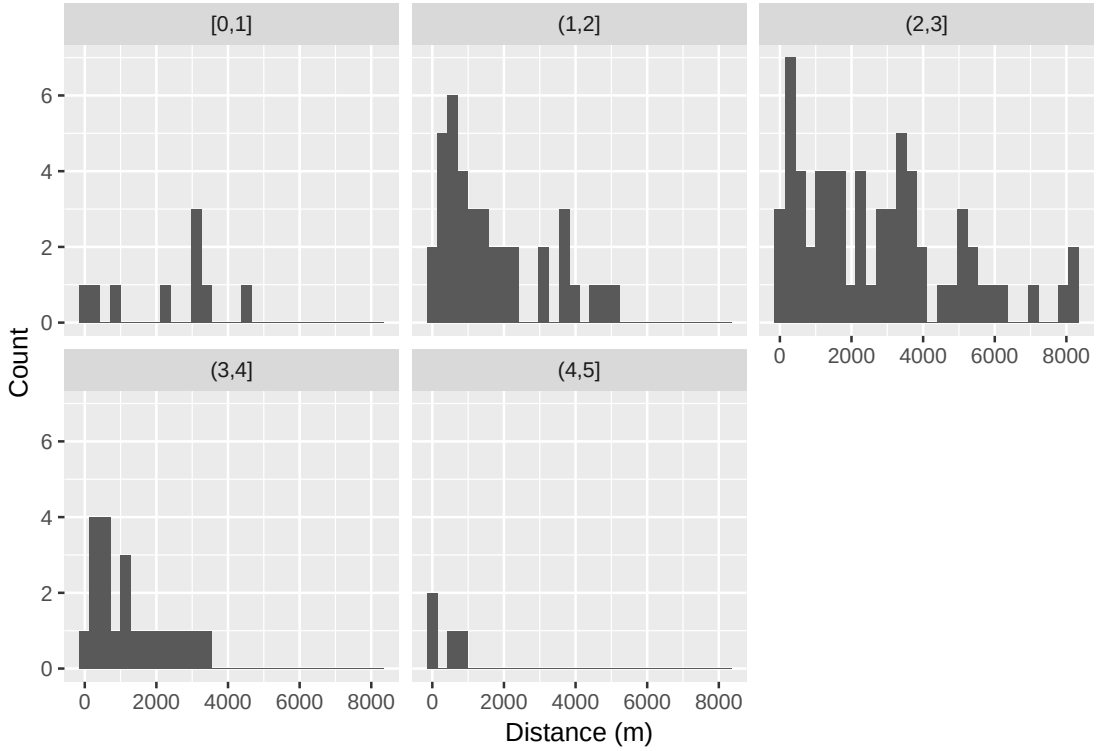


Figure 6.3: Effect of sea state upon detection distances.

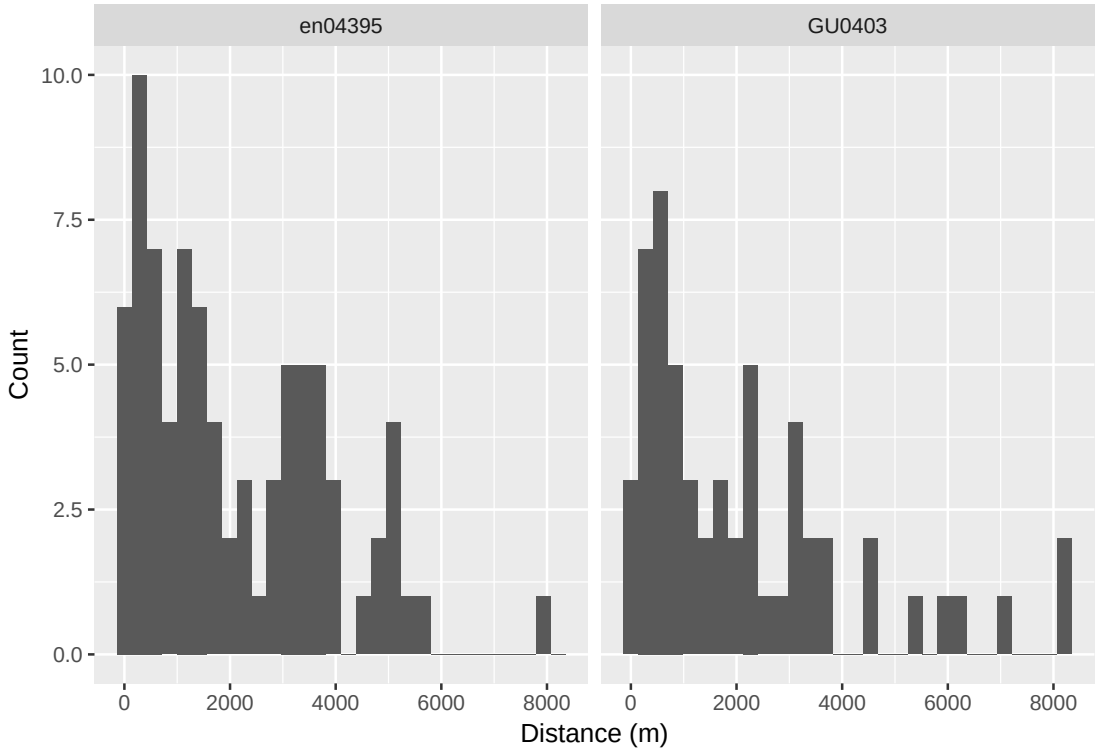


Figure 6.4: Effect of survey upon detection distances.

6.4.4 Survey effect

Given we are including data from two different surveys we can also investigate the relationship between survey and distances observed.

```
p <- ggplot(distdata) +
      geom_histogram(aes(distance)) +
      facet_wrap(~Survey) +
      labs(x="Distance (m)", y="Count")
print(p)
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

6.5 Fitting detection functions

It's now time to fit some detection function models. We'll use the `ds()` function from the `Distance` package to fit the detection function. You can access the help file for the `ds()` function by typing `?ds` – this will give you information about what the different arguments to the function are and what they do.

We can fit a very simple model with the following code:

```
df_hn <- ds(data=distdata, truncation=6000, key="hn", adjustment=NULL)
```

```
## Fitting half-normal key function
## Key only model: not constraining for monotonicity.
## AIC= 2252.06
## No survey area information supplied, only estimating detection function.
```

Let's dissect the call and see what each argument means:

- `data=distdata`: the data to use to fit the model, as we prepared above.
- `truncation=6000`: set the truncation distance. Here, observations at distances greater than 6000m will be discarded before fitting the detection function.
- `key="hn"`: the key function to use for the detection function, in this case half-normal (`?ds` lists the other options).
- `adjustment=NULL`: adjustment term series to fit. `NULL` here means that no adjustments should be fitted (again `?ds` lists all options).

Other useful arguments for this practical are:

- `formula=`: gives the formula to use for the scale parameter. By default it takes the value `~1`, meaning the scale parameter is constant and not a function of covariates.
- `order=`: specifies the "order" of the adjustments to be used. This is a number (or vector of numbers) specifying the order of the terms. For example `order=2` fits order 2 adjustments, `order=c(2,3)` will fit a model with order 2 and 3 adjustments (mathematically, it only makes sense to include order 3 with order 2). By default the value is `NULL` which has `ds()` select the number of adjustments by AIC.

6.5.1 Summaries

We can look at the summary of the fitted detection function using the `summary()` function:

```
summary(df_hn)
```

```
##
## Summary for distance analysis
## Number of observations :  132
## Distance range         :  0  -  6000
##
## Model : Half-normal key function
## AIC   : 2252.06
##
## Detection function parameters
## Scale coefficient(s):
##             estimate         se
## (Intercept) 7.900732 0.07884776
##
##                         Estimate          SE         CV
## Average p              0.5490484  0.03662569 0.06670757
## N in covered region 240.4159539 21.32287581 0.08869160
```

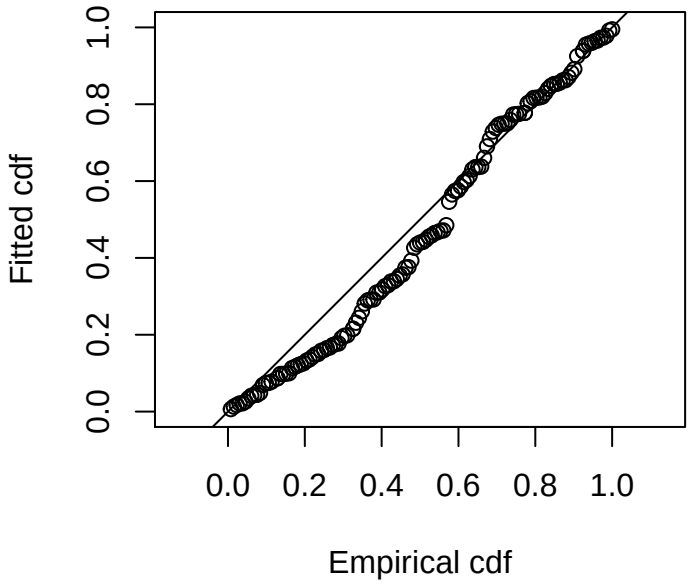


Figure 6.5: Goodness of fit QQ plot of half-normal detection function.

6.5.2 Goodness of fit

Goodness of fit quantile-quantile plot and test results can be accessed using the `ddf.gof()` function:

```
ddf.gof(df_hn$ddf)
```

```
##
## Goodness of fit results for ddf object
##
## Chi-square tests
##             [0,545] (545,1.09e+03] (1.09e+03,1.64e+03] (1.64e+03,2.18e+03]
## Observed  33.000000    20.00000000        19.000000000             8.000000
## Expected  21.708156    20.84245788        19.213254554            17.005089
## Chisquare  5.873634     0.03405238         0.002366986             4.768669
##           (2.18e+03,2.73e+03] (2.73e+03,3.27e+03] (3.27e+03,3.82e+03]
## Observed             9.000000          13.0000000           14.000000
## Expected            14.450499          11.7899695            9.235669
## Chisquare            2.055842           0.1241881            2.457737
##           (3.82e+03,4.36e+03] (4.36e+03,4.91e+03] (4.91e+03,5.45e+03]
## Observed             3.000000           4.0000000            7.000000
## Expected             6.946241           5.0159948            3.477683
## Chisquare            2.241906           0.2057908            3.567525
```

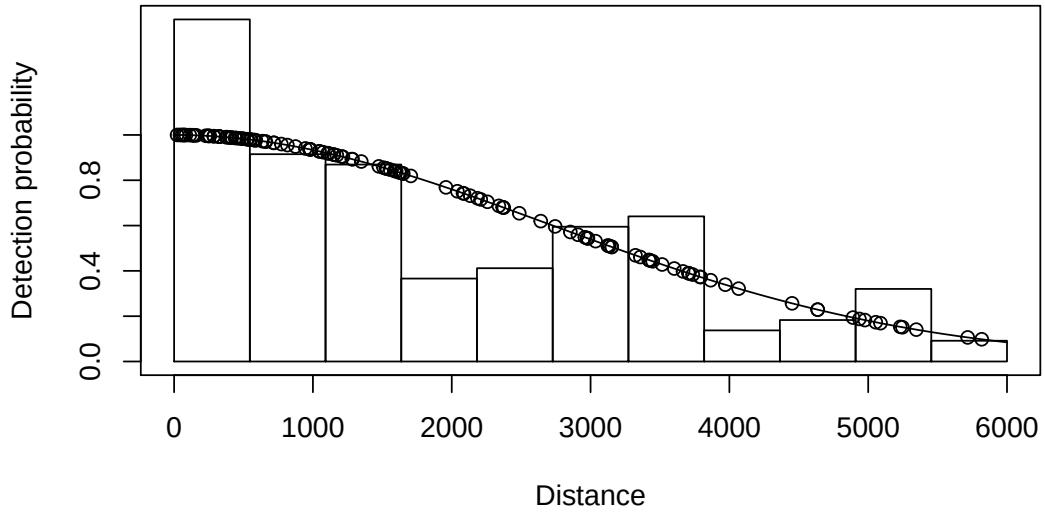


Figure 6.6: Half-normal detection function.

```
##           (5.45e+03,6e+03]     Total
## Observed        2.00000000 132.00000
## Expected        2.31498601 132.00000
## Chisquare       0.04285822  21.37457
##
## P = 0.011087 with 9 degrees of freedom
##
## Distance sampling Kolmogorov-Smirnov test
## Test statistic =  0.11192  P =  0.073241
##
## Distance sampling Cramer-von Mises test (unweighted)
## Test statistic =  0.39618  P =  0.073947
```

Note two things here: 1. We use the `$ddf` element of the detection function object 2. We're ignoring the χ^2 test results, as they rely on binning the distances to calculate test statistics where as Cramer-von Mises and Kolmogorov-Smirnov tests do not.

6.5.3 Plotting

We can plot the models simply using the `plot()` function:

```
plot(df_hn)
```

The dots on the plot indicate the distances where observations are. We can remove them (particularly

useful for a model without covariates) using the additional argument `showpoints=FALSE` (try this out!).

6.6 Now you try...

Now try fitting a few models and comparing them using AIC. Don't try to fit all possible models, just try a selection (say, a hazard-rate, a model with adjustments and a couple with different covariates). You can also try out changing the truncation distance.

Here's an example to work from. Some tips before you start:

- You can include as many lines as you like in a given chunk (though you may find it more manageable to separate them up, remembering each time to give the chunk a unique name).
- You can run the current line of code in RStudio by hitting `Control+Enter` (on Windows/Linux; `Command+Enter` on Mac).
- Giving your models informative names will help later on! Here I'm using `df_` to indicate that this is a detection function, then shortened forms of the model form and covariates, separated by underscores, but use what makes sense to you (and future you!).

```
df_hr_ss_size <- ds(distdata, truncation=6000, adjustment=NULL,
                    key="hr", formula=~SeaState+size)
```

```
## Fitting hazard-rate key function
```

```
## AIC= 2249.327
```

```
## No survey area information supplied, only estimating detection function.
```

Once you have the hang of writing models and looking at the differences between them, you should move onto the next section.

6.7 Model selection

Looking at the models individually can be a bit unwieldy – it's nicer to put that data into a table and sort the table by the relevant statistic. The function `summarize_ds_models()` in the `Distance` package performs this task for us.

The code below will make a results table with relevant statistics for model selection in it. The `summarize_ds_models()` function takes a series of object names as its first argument. We can do that with the two models that I fitted like so:

```
model_table <- summarize_ds_models(df_hn, df_hr_ss_size)
kable(model_table, digits=3, format="latex", booktabs=TRUE, row.names = FALSE, escape=FALSE,
      caption = "Comparison of half normal and hazard rate with sea state and group size.") %>
  kable_styling(latex_options="scale_down")
```

(You can add the models you fitted above into this list.)

Table 6.1: Comparison of half normal and hazard rate with sea state and group size.

Model	Key function	Formula	C-vM p-value	$\hat{P}_a$	se($\hat{P}_a$)	ΔAIC
`df_hr_ss_size`	Hazard-rate	SeaState + size	0.880	0.355	0.074	0.000
`df_hn`	Half-normal	1	0.074	0.549	0.037	2.733

6.7.0.1 A further note about model selection for the sperm whale data

Note that there is a considerable spike in our distance data. This may be down to observers guarding the trackline (spending too much time searching near zero distance). It's therefore possible that the hazard-rate model is overfitting to this peak. So we'd like to investigate results from the half-normal model too and see what the effects are on the final spatial models.

6.7.1 Estimating abundance

Just for fun, let's estimate abundance from these models using a Horvtiz-Thompson-type estimator.

We know the Horvitz-Thompson estimator has the following form:

$$\hat{N} = \frac{A}{a} \sum_{i=1}^{n} \frac{s_i}{p_i}$$

we can calculate each part of this equation in R:

- `A` is the total area of the region we want to estimate abundance for. This was $A = 5.285e+11m^2$.
- `a` is the total area we surveyed. We know that the total transect length was 9,498,474m and the truncation distance. Knowing that $a = 2wL$ we can calculate a.
- s_i are the group sizes, they are stored in `df_hn$ddf$data$size`.
- p_i are the probabilities of detection, we can obtain them using `predict(df_hn$ddf)$fitted`.

We know that in general operations are vectorised in R, so calculating `c(1, 2, 3)/c(4, 5, 6)` will give `c(1/4, 2/5, 3/6)` so we can just divide the results of getting the s_i and p_i values and then use the `sum()` function to sum them up.

Try out estimating abundance using the formula below using both `df_hn` and your favourite model from above:

Note that in the solutions to this exercise (posted on the course website) I show how to use the function `dht()` to estimate abundance (and uncertainty) for a detection function analysis. This involves some more complex data manipulation steps, so we've left it out here in favour of getting to grips with the mathematics.

6.7.1.1 Accounting for perception bias

It's common, especially in marine surveys, for animals at zero distance to be missed by observers. There are several ways to deal with this issue. For now, we are just going to use a very simply constant correction factor to inflate the abundance.

From Palka (2006), we think that observations on the track line were such that $g(0) = 0.46$, we can apply that correction to our abundance estimate (in a very primitive way):

This kind of correction works fine when we have a single number to adjust by, in general we'd like to model the perception bias using "mark-recapture distance sampling" techniques.

6.7.2 Save model objects

Save your top few models in an `RData` file, so we can load them up later on. We'll also save the distance data we used to fit out models.

```
save(df_hn, df_hr_ss_size, # add you models here, followed by commas!
     distdata,
     file="df-models.RData")
```

You can check it worked by using the `load()` function to recover the models.

Chapter 7

Simple density surface models

7.1 Aims

By the end of this practical, you should feel comfortable:

- Fitting a density surface model using `dsm()`
- Understanding what the objects that go into a `dsm()` call
- Understanding the role of the response in the `formula=` argument
- Understanding the output of `summary()` when called on a `dsm` object
- Increasing the `k` parameter of smooth terms to increase their flexibility
- Interpreting `gam.check` and `rqgam.check` plots and diagnostic output

The example code below uses the `df_hn` detection function in the density surface models. You can substitute this for your own best model as you go, or copy and paste the code at the end and see what results you get using your model for the detection function.

7.2 Load the packages and data

```
library(Distance)
library(dsm)
```

```
## Loading required package: numDeriv
```

```
## This is dsm 2.2.15
## Built: R 3.4.0; ; 2017-05-01 22:53:07 UTC; windows
```

```
library(ggplot2)
library(knitr)
```

Loading the `RData` files where we saved our results:

```
load("sperm-data.RData")
load("df-models.RData")
```

7.3 Pre-model fitting

Before we fit a model using `dsm()` we must first remove the observations from the spatial data that we excluded when we fitted the detection function – those observations at distances greater than the truncation.

```
obs <- obs[obs$distance <= df_hn$ddf$meta.data$width,]
```

Here we've used the value of the truncation stored in the detection function object, but we could also use the numeric value (which we can also find by checking the model's `summary()`).

Also note that if you want to fit DSMs using detection functions with different truncation distances, then you'll need to reload the `sperm-data.RData` and do the truncation again for that detection function.

7.4 Fitting DSMs

Using the data that we've saved so far, we can build a call to the `dsm()` function and fit out first density surface model. Here we're only going to look at models that include spatial smooths.

Let's start with a very simple model – a bivariate smooth of `x` and `y`:

```
dsm_nb_xy <- dsm(count~s(x,y),
                 ddf.obj=df_hn, segment.data = segs, observation.data=obs,
                 family=nb(), method="REML")
```

Note again that we try to have informative model object names so that we can work out what the main features of the model were from its name alone.

We can look at a `summary()` of this model. Look through the summary output and try to pick out the important information based on what we've talked about in the lectures so far.

```
summary(dsm_nb_xy)
```

```
##
## Family: Negative Binomial(0.105)
## Link function: log
##
## Formula:
## count ~ s(x, y) + offset(off.set)
##
## Parametric coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) -20.7009     0.2538  -81.56   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Approximate significance of smooth terms:
##          edf Ref.df Chi.sq  p-value
## s(x,y) 17.95  22.23  75.89 6.27e-08 ***
## ---
```

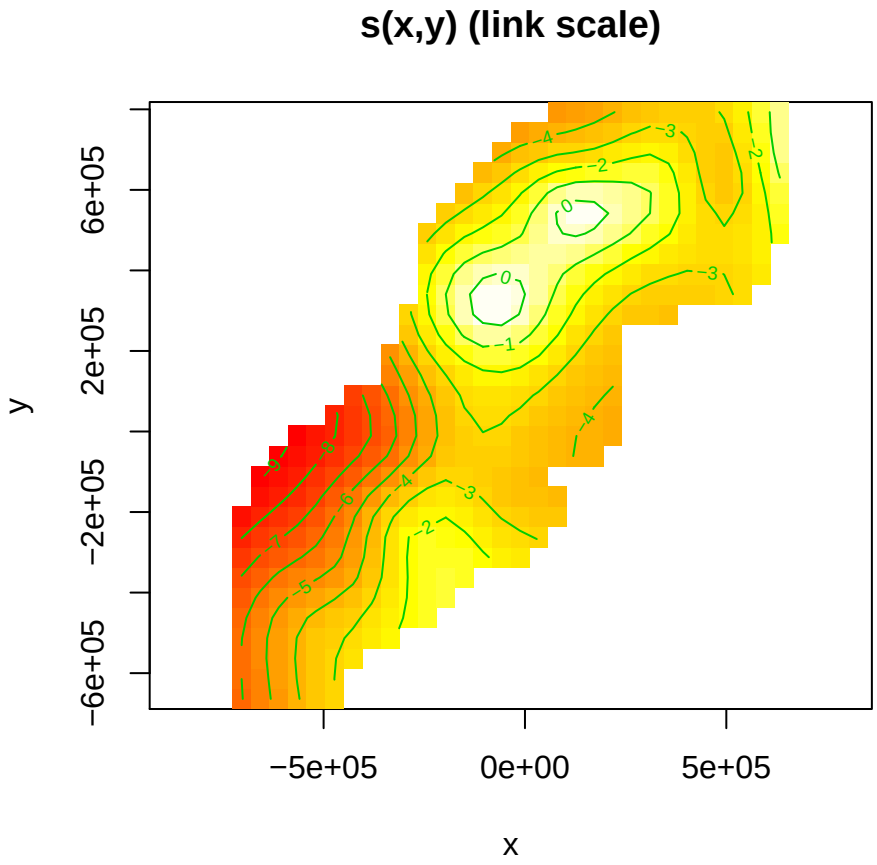


Figure 7.1: Fitted surface (on link scale) for s(x,y)

```
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## R-sq.(adj) =  0.0879   Deviance explained = 40.7%
## -REML = 392.65  Scale est. = 1         n = 949
```

7.4.1 Visualising output

As discussed in the lectures, the `plot` output is not terribly useful for bivariate smooths like these. We'll use `vis.gam()` to visualise the smooth instead:

```
vis.gam(dsm_nb_xy, view=c("x","y"), plot.type="contour",
        too.far=0.1, main="s(x,y) (link scale)", asp=1)
```

Notes:

1. The plot is on the scale of the link function, the offset is not taken into account – the contour values do not represent abundance, just the "influence" of the smooth.
2. We set `view=c("x","y")` to display the smooths for `x` and `y` (we can choose any two variables in our model to display like this)
3. `plot.type="contour"` gives this "flat" plot, set `plot.type="persp"` for a "perspective" plot, in 3D.
4. The `too.far=0.1` argument displays the values of the smooth not "too far" from the data (try changing this value to see what happens).
5. `asp=1` ensures that the aspect ratio of the plot is 1, making the pixels square.
6. Read the `?vis.gam` manual page for more information on the plotting options.

7.4.2 Checking the model

We can use the `gam.check()` and `rqgam.check` functions to check the model.

```
gam.check(dsm_nb_xy)
```

```
##
## Method: REML   Optimizer: outer newton
## full convergence after 5 iterations.
## Gradient range [-4.081497e-08,5.889688e-08]
## (score 392.646 & scale 1).
## Hessian positive definite, eigenvalue range [2.157927,29.21001].
## Model rank =  30 / 30
##
## Basis dimension (k) checking results. Low p-value (k-index<1) may
## indicate that k is too low, especially if edf is close to k'.
##
##        k' edf k-index p-value
## s(x,y) 29  18    0.53  <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
rqgam.check(dsm_nb_xy, pch=20)
```

Remember that the left side of the `gam.check()` plot and the right side of the `rqgam.check()` plot are most useful.

Looking back through the lecture notes, do you see any problems in these plots or in the text output from `gam.check()`.

7.4.3 Setting basis complexity

We can set the basis complexity via the `k` argument to the `s()` term in the formula. For example the following re-fits the above model with a much smaller basis complexity than before:

```
dsm_nb_xy_smallk <- dsm(count~s(x, y, k=10),
                        ddf.obj=df_hn, segment.data = segs, observation.data=obs,
                        family=nb(), method="REML")
```

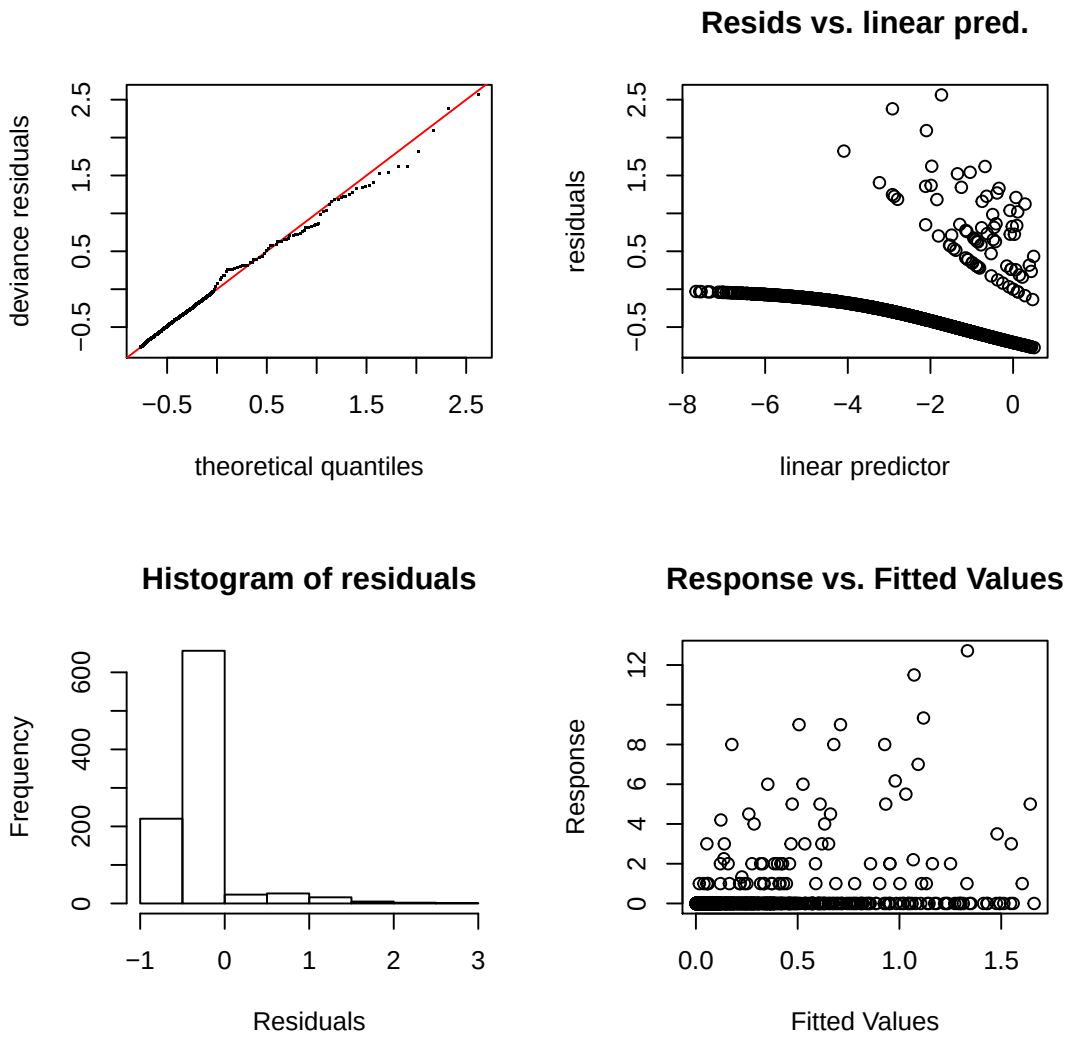


Figure 7.2: Gam check results s(x,y) neg-binomial.

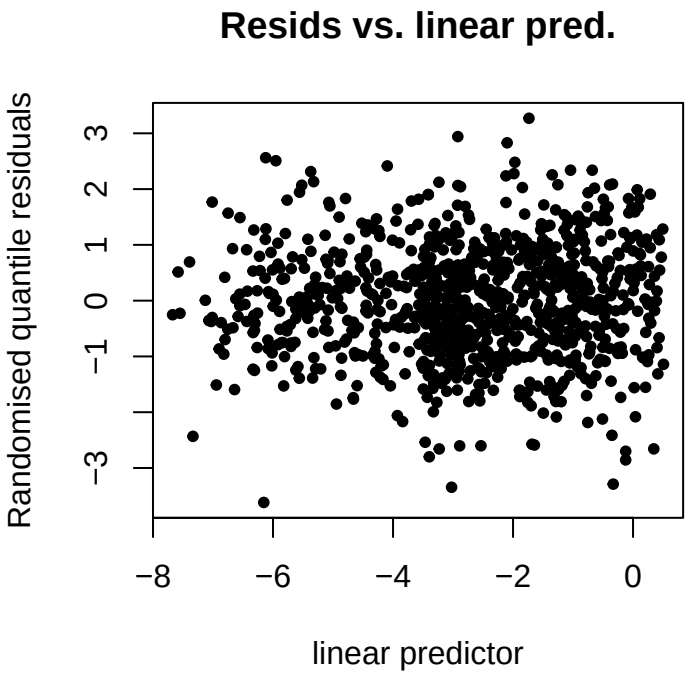


Figure 7.3: Residual quartile gam check results s(x,y) neg-binomial.

Compare the output of `vis.gam()` and `gam.check()` for this model to the model with a larger basis complexity.

7.5 Estimated abundance as response

So far we've just used `count` as the response. That is, we adjusted the offset of the model to make it take into account the "effective area" of the segments (see lecture notes for a refresher).

Instead of using `count` we could use `abundance.est`, which will leave the segment areas as they are and calculate the Horvitz-Thompson estimates of the abundance per segment and use that as the response in the model. This is most useful when we have covariates in the detection function (though we can use it any time).

Try copying the code that fits the model `dsm_nb_xy` and make a model `dsm_nb_xy_ae` that replaces `count` for `abundance.est` in the model formula and uses the `df_hr_ss_size` detection function. Compare the results of summaries, plots and checks between this and the count model.

7.6 Univariate models

Instead of fitting a bivariate smooth of `x` and `y` using `s(x, y)`, we could instead use the additive nature and fit the following model:

```
dsm_nb_x_y <- dsm(count~s(x)+ s(y),
                  ddf.obj=df_hn, segment.data = segs, observation.data=obs,
                  family=nb(), method="REML")
```

Compare this model with `dsm_nb_xy` using `vis.gam()` (Note you can display two plots side-by-side using `par(mfrow=c(1,2))`). Investigate the output from `summary()` and the check functions too, comparing with the other models, adjust `k` if necessary.

7.7 Tweedie response distribution

So far, we've used `nb()` as the response – the negative binomial distribution. We can also try out the Tweedie distribution as a response by replacing `nb()` with `tw()`.

Try this out and compare the resulting check plots.

7.8 Save models

It'll be interesting to see how these models compare to the more complex models we'll see later on. Let's save the fitted models at this stage.

```
# add your models here
save(dsm_nb_x_y, dsm_nb_xy,
     file="dsms-xy.RData")
```

7.9 Extra credit

If you have time, try the following:

- What happens when we set `family=quasipoisson()`? Compare results of `gam.check` and `rqgam.check` for this and the other models.
- Make the `k` value very big (~100 or so), what do you notice?

Chapter 8

Advanced density surface models

8.1 Aims

By the end of this practical, you should feel comfortable:

- Fitting DSMs with multiple smooth terms in them
- Selecting smooth terms by p-values
- Using shrinkage smoothers
- Selecting between models using deviance, REML score
- Investigating concurvity in DSMs with multiple smooths
- Investigating sensitivity sensitivity and path dependence

8.2 Load data and packages

```
library(Distance)
library(dsm)
library(ggplot2)
library(knitr)
library(kableExtra)
library(plyr)
```

```
##
## Attaching package: 'plyr'
```

```
## The following object is masked from 'package:secr':
##
##     join
```

```
library(reshape2)
```

Loading the data processed from GIS and the fitted detection function objects from the previous exercises:

```
load("sperm-data.RData")
load("df-models.RData")
```

8.3 Exploratory analysis

We can do some exploratory analysis by aggregating the counts to each cell and plotting what's going on.

Don't worry about understanding what this code is doing at the moment.

```
# join the observations onto the segments
join_dat <- join(segs, obs, by="Sample.Label", type="full")
# sum up the observations per segment
n <- ddply(join_dat, .(Sample.Label), summarise, n=sum(size), .drop = FALSE)
# sort the segments by their labsl
segs_eda <- segs[sort(segs$Sample.Label),]
# make a new column for the counts
segs_eda$n <- n$n

# remove the columns we don't need,
segs_eda$CentreTime <- NULL
segs_eda$POINT_X <- NULL
segs_eda$POINT_Y <- NULL
segs_eda$segment.area <- NULL
segs_eda$off.set <- NULL
segs_eda$CenterTime <- NULL
segs_eda$Effort <- NULL
segs_eda$Length <- NULL
segs_eda$SegmentID <- NULL
segs_eda$coords.x1 <- NULL
segs_eda$coords.x2 <- NULL

# "melt" the data so we have four columns:
#   Sample.Label, n (number of observations),
#   variable (which variable), value (its value)
segs_eda <- melt(segs_eda, id.vars=c("Sample.Label", "n"))
# try head(segs_eda)
```

Finally, we can plot histograms of counts for different values of the covariates:

```
p <- ggplot(segs_eda) +
       geom_histogram(aes(value, weight=n)) +
       facet_wrap(~variable, scale="free") +
       xlab("Covariate value") +
       ylab("Aggregated counts")
```

```
## Warning: Ignoring unknown aesthetics: weight
```

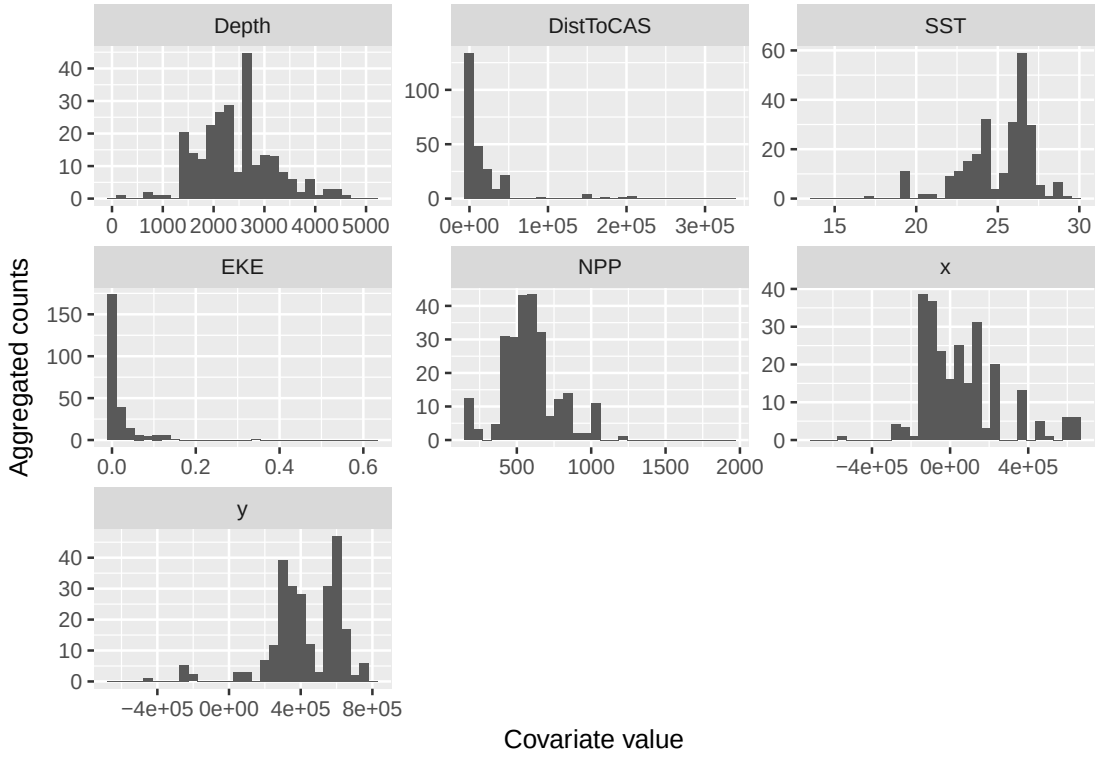


Figure 8.1: Histograms of segment counts at various covariate levels.

```
print(p)
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

We can also just plot the counts against the covariates, note the high number of zeros (but still some interesting patterns):

```
p <- ggplot(segs_eda) +
       geom_point(aes(value, n)) +
       facet_wrap(~variable, scale="free") +
       xlab("Covariate value") +
       ylab("Aggregated counts")
print(p)
```

```
## Warning: Removed 6076 rows containing missing values (geom_point).
```

These plots give a very rough idea of the relationships we can expect in the model. Notably these plots don't take into account interactions between the variables and potential correlations between the terms, as well as detectability.

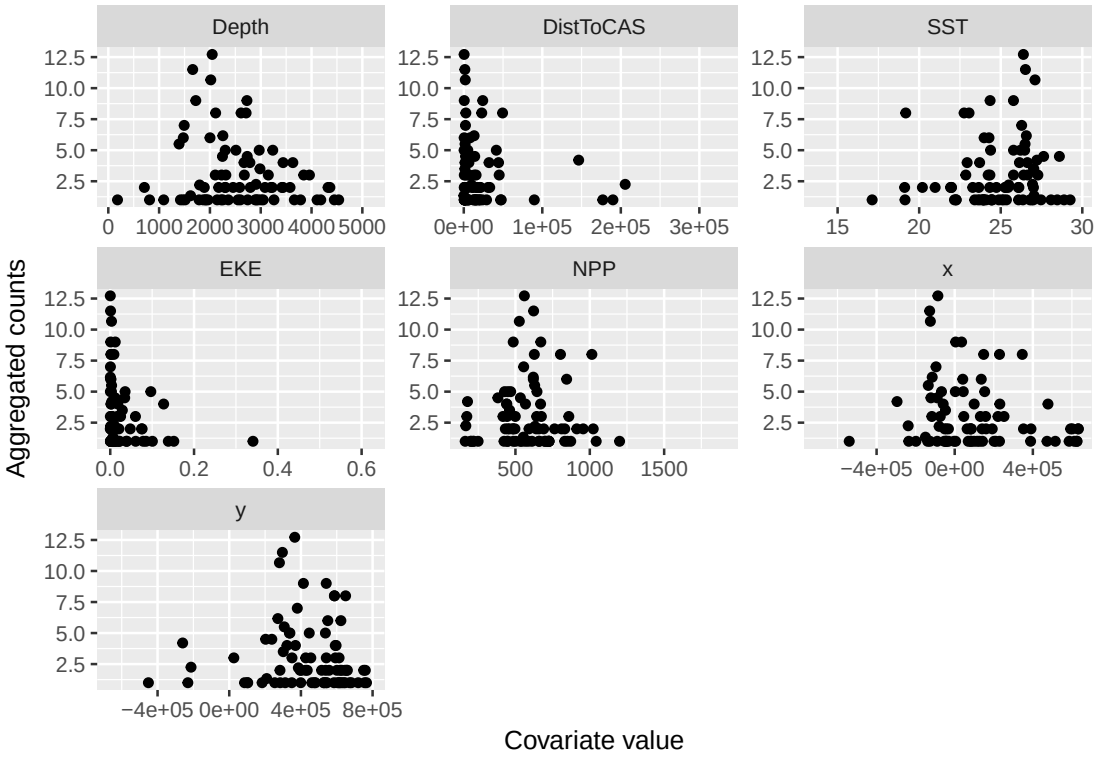


Figure 8.2: Relationship of segment counts to covariate values.

8.4 Pre-model fitting

As we did in the previous exercise we must remove the observations from the spatial data that we excluded when we fitted the detection function – those observations at distances greater than the truncation.

```
obs <- obs[obs$distance <= df_hn$ddf$meta.data$width,]
```

Here we've used the value of the truncation stored in the detection function object, but we could also use the numeric value (which we can also find by checking the model's `summary()`).

Again note that if you want to fit DSMs using detection functions with different truncation distances, then you'll need to reload the `sperm-data.RData` and do the truncation again for that detection function.

8.5 Our new friend +

We can build a really big model using + to include all the terms that we want in the model. We can check what's available to us by using `head()` to look at the segment table:

```
head(segs)
```

```
##            CenterTime SegmentID   Length  POINT_X  POINT_Y     Depth
## 1 2004/06/24 07:27:04         1 10288.91 214544.0 689074.3  118.5027
## 2 2004/06/24 08:08:04         2 10288.91 222654.3 682781.0  119.4853
## 3 2004/06/24 09:03:18         3 10288.91 230279.9 675473.3  177.2779
## 4 2004/06/24 09:51:27         4 10288.91 239328.9 666646.3  527.9562
## 5 2004/06/24 10:25:39         5 10288.91 246686.5 659459.2  602.6378
## 6 2004/06/24 11:00:22         6 10288.91 254307.0 652547.2 1094.4402
##    DistToCAS      SST          EKE      NPP coords.x1 coords.x2        x
## 1 14468.1533 15.54390 0.0014442616 1908.129  214544.0  689074.3 214544.0
## 2 10262.9648 15.88358 0.0014198086 1889.540  222654.3  682781.0 222654.3
## 3  6900.9829 16.21920 0.0011704842 1842.057  230279.9  675473.3 230279.9
## 4  1055.4124 16.45468 0.0004101589 1823.942  239328.9  666646.3 239328.9
## 5  1112.6293 16.62554 0.0002553244 1721.949  246686.5  659459.2 246686.5
## 6   707.5795 16.83725 0.0006556266 1400.281  254307.0  652547.2 254307.0
##          y   Effort Sample.Label
## 1 689074.3 10288.91            1
## 2 682781.0 10288.91            2
## 3 675473.3 10288.91            3
## 4 666646.3 10288.91            4
## 5 659459.2 10288.91            5
## 6 652547.2 10288.91            6
```

We can then fit a model with the available covariates in it, each as an `s()` term.

```
dsm_nb_xy_ms <- dsm(count~s(x,y, bs="ts") +
                      s(Depth, bs="ts") +
                      s(DistToCAS, bs="ts") +
                      s(SST, bs="ts") +
```

```
                        s(EKE, bs="ts") +
                        s(NPP, bs="ts"),
                  df_hn, segs, obs,
                  family=nb(), method="REML")
summary(dsm_nb_xy_ms)
```

```
##
## Family: Negative Binomial(0.114)
## Link function: log
##
## Formula:
## count ~ s(x, y, bs = "ts") + s(Depth, bs = "ts") + s(DistToCAS,
##     bs = "ts") + s(SST, bs = "ts") + s(EKE, bs = "ts") + s(NPP,
##     bs = "ts") + offset(off.set)
##
## Parametric coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) -20.7732     0.2295   -90.5   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Approximate significance of smooth terms:
##                     edf Ref.df Chi.sq  p-value
## s(x,y)        1.8636924     29 19.141 2.90e-05 ***
## s(Depth)      3.4176460      9 46.263 1.65e-11 ***
## s(DistToCAS) 0.0000801      9  0.000   0.9053
## s(SST)        0.0002076      9  0.000   0.5402
## s(EKE)        0.8563344      9  5.172   0.0134 *
## s(NPP)        0.0001018      9  0.000   0.7820
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## R-sq.(adj) =  0.0947   Deviance explained = 39.2%
## -REML = 382.76  Scale est. = 1         n = 949
```

Notes:

1. We're using `bs="ts"` to use the shrinkage thin plate regression spline. More technical detail on these smooths can be found on their manual page `?smooth.construct.ts.smooth.spec`.
2. We've not specified basis complexity (`k`) at the moment. Note that if you want to specify the same complexity for multiple terms, it's often easier to make a variable that can then be given as `k` (for example, setting `k1<-15` and then setting `k=k1` in the required `s()` terms).

8.5.1 Plot

Let's plot the smooths from this model:

```
plot(dsm_nb_xy_ms, pages=1, scale=0)
```

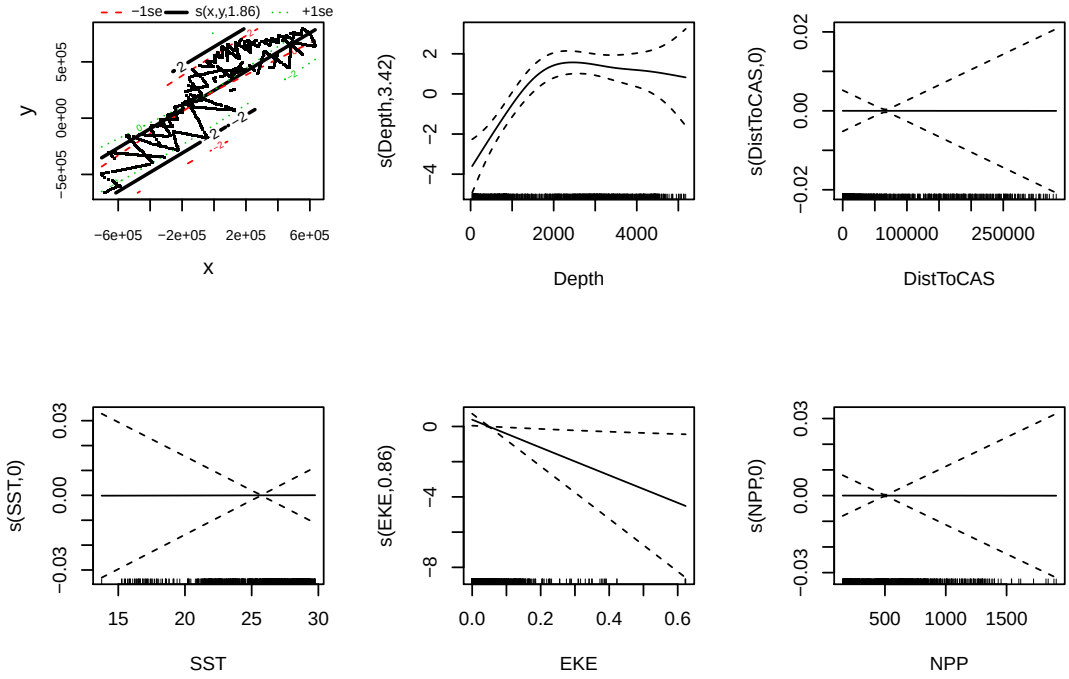


Figure 8.3: Smooths for all covariates with neg-binomial response distribution.

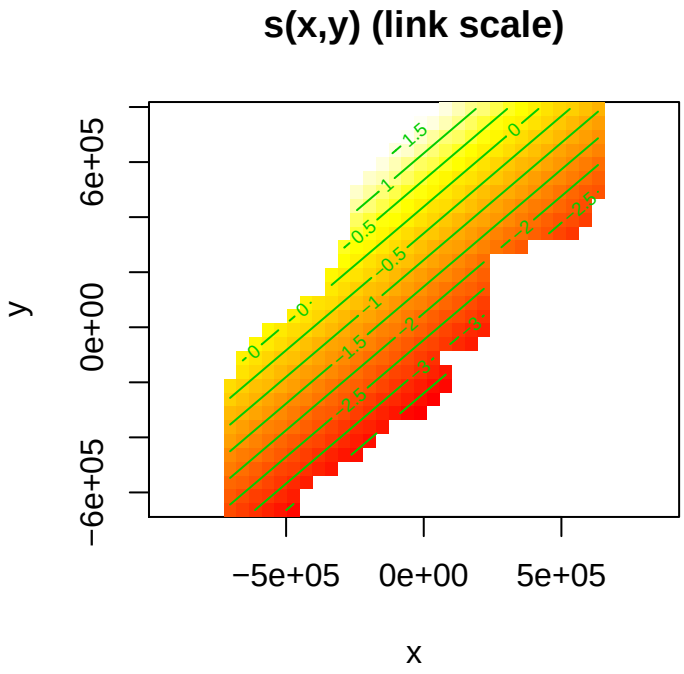


Figure 8.4: Fitted surface with all environmental covariates, and neg-binomial response distribution.

Notes:

1. Setting `shade=TRUE` gives prettier confidence bands.
2. As with `vis.gam()` the response is on the link scale.
3. `scale=0` puts each plot on a different y-axis scale, making it easier to see the effects. Setting `scale=-1` will put the plots on a common y-axis scale

We can also plot the bivariate smooth of `x` and `y` as we did before, using `vis.gam()`:

```
vis.gam(dsm_nb_xy_ms, view=c("x","y"), plot.type="contour", too.far=0.1,
        main="s(x,y) (link scale)", asp=1)
```

Compare this plot to Figure 7.1, generated in the previous practical when only `x` and `y` were included in the model.

8.5.2 Check

As before, we can use `gam.check()` and `rqgam.check()` to look at the residual check plots for this model. Do this in the below gaps and comment on the resulting plots and diagnostics.

You might decide from the diagnostics that you need to increase `k` for some of the terms in the model. Do this and re-run the above code to ensure that the smooths are flexible enough. The `?choose.k`

manual page can offer some guidance. Generally if the EDF is close to the value of `k` you supplied, it's worth doubling `k` and refitting to see what happens. You can always switch back to the smaller `k` if there is little difference.

8.5.3 Select terms

As was covered in the lectures, we can select terms by (approximate) p-values and by looking for terms that have EDFs significantly less than 1 (those which have been shrunk).

Decide on a significance level that you'll use to discard terms in the model. Remove the terms that are non-significant at this level and re-run the above checks, summaries and plots to see what happens. It's helpful to make notes to yourself as you go

It's easiest to either comment out the terms that are to be removed (using #) or by copying the code chunk above and pasting it below.

Having removed a smooth and reviewed your model, you may decide you wish to remove another. Follow the process again, removing a term, looking at plots and diagnostics.

8.5.4 Compare response distributions

Use the `gam.check()` to compare quantile-quantile plots between negative binomial and Tweedie distributions for the response.

8.6 Estimated abundance as a response

Again, we've only looked at models with `count` as the response. Try using a detection function with covariates and the `abundance.est` response in the chunk below:

8.7 Concurvity

Checking concurvity (Amodio, Aria, and D'Ambrosio (2014)) of terms in the model can be accomplished using the `concurvity()` function.

```
concurvity(dsm_nb_xy_ms)
```

```
##                  para    s(x,y)  s(Depth) s(DistToCAS)    s(SST)    s(EKE)
## worst    9.804613e-24 0.9963493 0.9836597    0.9959057 0.9772853 0.7702479
## observed 9.804613e-24 0.8597372 0.8277050    0.9879372 0.9523512 0.6746585
## estimate 9.804613e-24 0.7580838 0.9272203    0.9642030 0.8978412 0.4906765
##             s(NPP)
## worst    0.9727752
## observed 0.9525363
## estimate 0.8694619
```

By default the function returns a matrix of a measure of concurvity between one of the terms and the rest of the model.

Compare the output of the models before and after removing terms.

Reading these matrices can be laborious and not very fun. The function `vis.concurvity()` in the `dsm` package is used to visualise the concurvity *between terms* in a model by colour coding the matrix (and blanking out the redundant information).

Again compare the results of plotting for models with different terms.

8.8 Sensitivity

8.8.1 Compare bivariate and additive spatial effects

If we replace the bivariate smooth of location (`s(x, y)`) with an additive terms (`s(x)+s(y)`), we may see a difference in the final model (different covariates selected).

```
dsm_nb_x_y_ms <- dsm(count~s(x, bs="ts") +
                         s(y, bs="ts") +
                         s(Depth, bs="ts") +
                         s(DistToCAS, bs="ts") +
                         s(SST, bs="ts") +
                         s(EKE, bs="ts") +
                         s(NPP, bs="ts"),
                     df_hn, segs, obs,
                     family=nb(), method="REML")
summary(dsm_nb_x_y_ms)
```

```
##
## Family: Negative Binomial(0.116)
## Link function: log
##
## Formula:
## count ~ s(x, bs = "ts") + s(y, bs = "ts") + s(Depth, bs = "ts") +
##     s(DistToCAS, bs = "ts") + s(SST, bs = "ts") + s(EKE, bs = "ts") +
##     s(NPP, bs = "ts") + offset(off.set)
##
## Parametric coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) -20.7743     0.2274  -91.37   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Approximate significance of smooth terms:
##                    edf Ref.df Chi.sq  p-value
## s(x)         2.875e-01      9  0.337   0.2698
## s(y)         1.709e-05      9  0.000   0.6304
## s(Depth)     3.391e+00      9 37.216 9.88e-10 ***
```

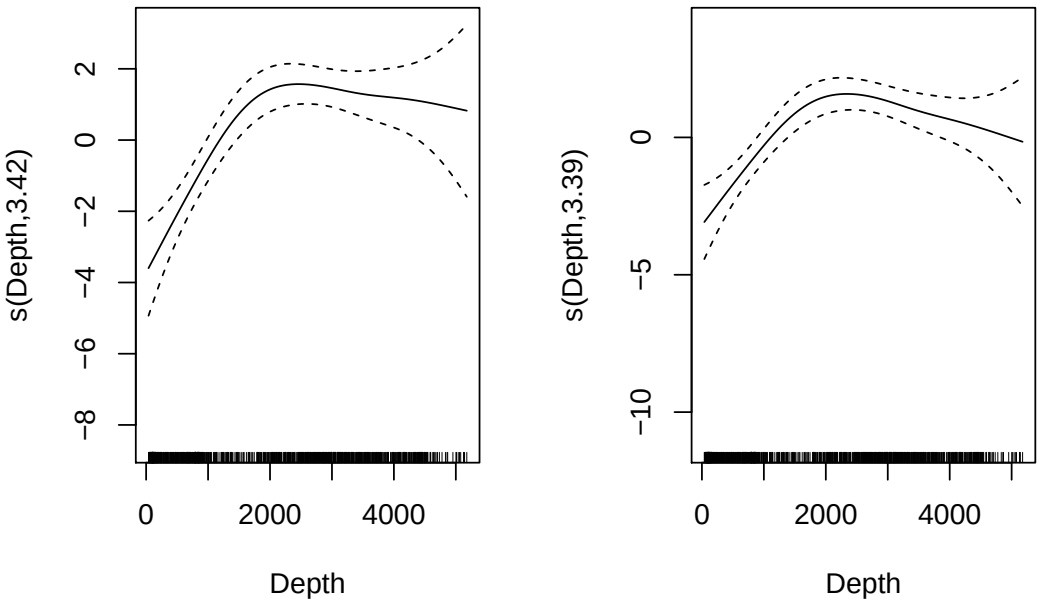


Figure 8.5: Shape of depth covariate response with bivariate s(x,y) and univariate s(x)+s(y).

```
## s(DistToCAS) 5.393e-04      9  0.000   0.5246
## s(SST)       9.814e-05      9  0.000   0.8176
## s(EKE)       8.670e-01      9  5.582   0.0103 *
## s(NPP)       2.844e+00      9 23.236 9.13e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## R-sq.(adj) =  0.0993   Deviance explained = 40.7%
## -REML = 383.03  Scale est. = 1         n = 949
```

Try performing model selection as before from this base model and compare the resulting models.

Compare the resulting smooths from like terms in the model. For example, if depth were selected in both models, compare EDFs and plots, e.g.:

```
par(mfrow=c(1,2))
plot(dsm_nb_xy_ms, select=2)
plot(dsm_nb_x_y_ms, select=3)
```

Note that there `select=` picks just one term to plot. These are in the order in which the terms occur in the `summary()` output (so you may well need to adjust the above code).

Table 8.1: Model performance of s(x,y) and s(x)+s(y) in presence of other covariates.

	response	terms	AIC	REML	Deviance_explained
dsm_nb_x_y_ms	Negative Binomial(0.116)	s(x), s(y), s(Depth), s(DistToCAS), s(SST), s(EKE), s(NPP)	752.5585	383.0326	40.73%
dsm_nb_xy_ms	Negative Binomial(0.114)	s(x,y), s(Depth), s(DistToCAS), s(SST), s(EKE), s(NPP)	754.0326	382.7591	39.2%

8.9 Comparing models

As with the detection functions in the earlier exercises, here is a quick function to generate model results tables with appropriate summary statistics:

```
summarize_dsm <- function(model){

  summ <- summary(model)

  data.frame(response = model$family$family,
             terms    = paste(rownames(summ$s.table), collapse=", "),
             AIC      = AIC(model),
             REML     = model$gcv.ubre,
             "Deviance_explained" = paste0(round(summ$dev.expl*100,2),"%")
            )

}
```

We can, again, make a list of the models and give that to the above function

```
# add your models to this list!
model_list <- list(dsm_nb_x_y_ms, dsm_nb_xy_ms)
library(plyr)
summary_table <- ldply(model_list, summarize_dsm)
row.names(summary_table) <- c("dsm_nb_x_y_ms", "dsm_nb_xy_ms")
```

```
summary_table <- summary_table[order(summary_table$REML, decreasing=TRUE),]
kable(summary_table,
      caption = "Model performance of s(x,y) and s(x)+s(y) in presence of other covariates.")
  kable_styling(latex_options="scale_down")
```

8.10 Saving models

Now save the models that you'd like to use to predict with later. I recommend saving as many models as you can so you can compare their results in the next practical.

```
# add your models here
save(dsm_nb_xy_ms, dsm_nb_x_y_ms,
     file="dsms.RData")
```

Chapter 9

Prediction using fitted density surface models

Now we've fitted some models, let's use the `predict` functions and the data from GIS to make predictions of abundance.

9.1 Aims

By the end of this practical, you should feel comfortable:

- Loading raster data into R
- Building a `data.frame` of prediction covariates
- Making a prediction using the `predict()` function
- Summing the prediction cells to obtain a total abundance for a given area
- Plotting a map of predictions
- Saving predictions to a raster to be used in ArcGIS

9.2 Loading the packages and data

```
library(knitr)
library(dsm)
library(ggplot2)
# colourblind-friendly colourschemes
library(viridis)
```

```
## Loading required package: viridisLite
```

```
# to load and save raster data
library(raster)
```

```
##
## Attaching package: 'raster'

## The following object is masked from 'package:nlme':
##
##     getData

## The following objects are masked from 'package:secr':
##
##     flip, rotate, shift, trim
```

```
# models with only spatial terms
load("dsms-xy.RData")
# models with all covariates
load("dsms.RData")
```

9.3 Loading prediction data

Before we can make predictions we first need to load the covariates into a "stack" from their files on disk using the `stack()` function from `raster`. We give `stack()` a vector of locations to load the rasters from. Note that in RStudio you can use tab-completion for these locations and avoid some typing.At this point we arbitrarily choose the time periods of the SST, NPP and EKE rasters (2 June 2004, or Julian date 153).

```
predictorStack <- stack(c("../../spermwhale-analysis/Covariates_for_Study_Area/Depth.img",
                          "../../spermwhale-analysis/Covariates_for_Study_Area/GLOB/CMC/CMC0.2
                          "../../spermwhale-analysis/Covariates_for_Study_Area/VGPM/Rasters/vg
                          "../../spermwhale-analysis/Covariates_for_Study_Area/DistToCanyonsAr
                          "../../spermwhale-analysis/Covariates_for_Study_Area/Global/DT\ all\
                          ))
```

We need to rename the layers in our stack to match those in the model we are going to use to predict. If you need a refresher on the names that were used there, call `summary()` on the DSM object.

```
names(predictorStack) <- c("Depth","SST","NPP", "DistToCAS", "EKE")
```

Now these are loaded, we can coerce the stack into something `dsm` can talk to using the `as.data.frame` function. Note we need the `xy=TRUE` to ensure that `x` and `y` are included in the prediction data. We also set the offset value – the area of each cell in our prediction grid.

```
predgrid <- as.data.frame(predictorStack, xy=TRUE)
predgrid$off.set <- (10*1000)^2
```

We can then predict for the model `dsm_nb_xy_ms`:

```
pp <- predict(dsm_nb_xy_ms, predgrid)
```

This is just a list of numbers – the predicted abundance per cell. We can sum these to get the estimated abundance for the study area:

```
sum(pp, na.rm=TRUE)
```

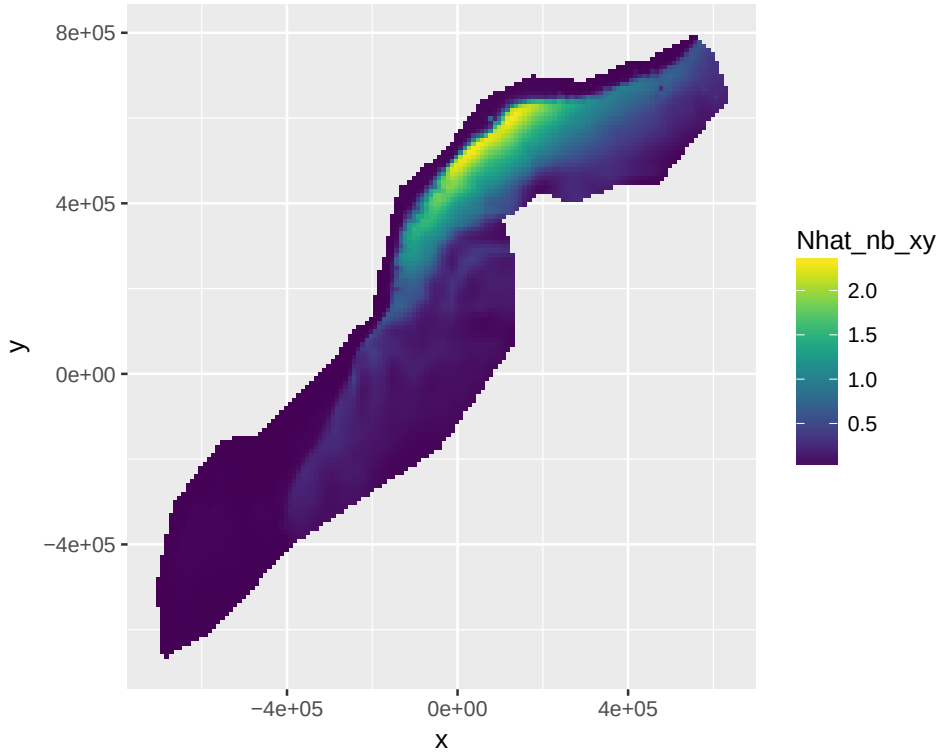


Figure 9.1: Predicted surface for abundance estimates with bivariate spatial smooth along with environmental covariates.

```
## [1] 1710.347
```

Because we predicted over the whole raster grid (including those cells without covariate values – e.g. land), some of the values in `pp` will be `NA`, so we can ignore them when we sum by setting `na.rm=TRUE`. We need to do this again when we plot the data too.

We can also plot this to get a spatial representation of the predictions:

```
# assign the predictions to the prediction grid data.frame
predgrid$Nhat_nb_xy <- pp
# remove the NA entries (because of the grid structure of the raster)
predgrid_plot <- predgrid[!is.na(predgrid$Nhat_nb_xy),]
# plot!
p <- ggplot(predgrid_plot) +
      geom_tile(aes(x=x, y=y, fill=Nhat_nb_xy, width=10*1000, height=10*1000)) +
      coord_equal() +
      scale_fill_viridis()
```

```
## Warning: Ignoring unknown aesthetics: width, height
```

```
print(p)
```

Copy the chunk above and make predictions for the other models you saved in the previous exercises.

In particular, compare the models with only spatial terms to those with environmental covariates included.

9.4 Save the prediction to a raster

To be able to load our predictions into ArcGIS, we need to save them as a raster file. First we need to make our predictions into a raster object and save them to the stack we already have:

```
# setup the storage for the predictions
pp_raster <- raster(predictorStack)
# put the values in, making sure they are numeric first
pp_raster <- setValues(pp_raster, as.numeric(pp))
# name the new, last, layer in the stack
names(pp_raster) <- "Nhat_nb_xy"
```

We can then save that object to disk as a raster file:

```
writeRaster(pp_raster, "abundance_raster.img", datatype="FLT4S", overwrite=TRUE)
```

Here we just saved one raster layer: the predictions from model `Nhat_nb_xy`. Try saving another set of predictions from another model by copying the above chunk.

You can check that the raster was written correctly by using the `stack()` function, as we did before to load the data and then the `plot()` function to see what was saved in the raster file.

9.5 Save prediction grid to `RData`

We'll need to use the prediction grid and predictor stack again when we calculate uncertainty in the next practical, so let's save those objects now to save time later.

```
save(predgrid, predictorStack, file="predgrid.RData")
```

9.6 Extra credit

- Try refitting your models with `family=quasipoisson()` as the response distribution. What do you notice about the predicted abundance?
- Can you work out a way to use `ldply()` from the `plyr` package so that you can use `facet_wrap` in `ggplot2` to plot predictions for multiple models in a grid layout?

Chapter 10

Estimating precision of predictions from density surface models

Now we've fitted some models and estimated abundance, we can estimate the variance associated with the abundance estimate (and plot it).

10.1 Aims

By the end of this practical, you should feel comfortable:

- Knowing when to use `dsm.var.prop` and when to use `dsm.var.gam`
- Estimating variance for a given prediction area
- Estimating variance per-cell for a prediction grid
- Interpreting the `summary()` output for uncertainty estimates
- Making maps of the coefficient of variation in R
- Saving uncertainty information to a raster file to be read by ArcGIS

10.2 Load packages and data

```
library(dsm)
library(raster)
library(ggplot2)
library(viridis)
library(plyr)
library(knitr)
library(kableExtra)
library(rgdal)
```

Load the models and prediction grid:

```
load("dsms.RData")
load("dsms-xy.RData")
load("predgrid.RData")
```

10.3 Estimation of variance

Depending on the model response (count or Horvitz-Thompson) we can use either `dsm.var.prop` or `dsm.var.gam`, respectively. `dsm_nb_xy_ms` doesn't include any covariates at the observer level in the detection function, so we can use the variance propagation method and estimate the uncertainty in detection function parameters in one step.

```
# need to remove the NAs as we did when plotting
predgrid_var <- predgrid[!is.na(predgrid$Depth),]
# now estimate variance
var_nb_xy_ms <- dsm.var.prop(dsm_nb_xy_ms, predgrid_var,
                             off.set=predgrid_var$off.set)
```

To summarise the results of this variance estimate:

```
summary(var_nb_xy_ms)
```

```
## Summary of uncertainty in a density surface model calculated
##  by variance propagation.
##
## Quantiles of differences between fitted model and variance model
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -1.31031 -0.04789  0.17085  0.01322  0.23876  0.27258
##
## Approximate asymptotic confidence interval:
##         2.5%         Mean        97.5%
##     29.57459   2015.81785 137399.06890
## (Using log-Normal approximation)
##
## Point estimate                 : 2015.818
## Standard error                 : 20412.4
## Coefficient of variation       : 10.1261
```

Try this out for some of the other models you've saved. Remember to use `dsm.var.gam` when there are covariates in the detection function and `dsm.var.prop` when there aren't.

10.4 Summarise multiple models

We can again summarise all the models, as we did with the DSMs and detection functions, now including the variance:

```
summarize_dsm_var <- function(model, predgrid){
```

Table 10.1: Model performance: bivariate vs univariate spatial smooths without and with environmental covariates.

.id	response	terms	AIC	REML	Deviance_explained	lower_CI	Nhat	upper_CI
dsm_nb_xy	Negative Binomial(0.105)	s(x,y)	775.3	392.6	40.65%	21.2	1529.0	110244.1
dsm_nb_x_y	Negative Binomial(0.085)	s(x), s(y)	789.8	395.9	31.14%	21.2	1529.0	110244.1
dsm_nb_xy_ms	Negative Binomial(0.114)	s(x,y), s(Depth), s(DistToCAS), s(SST), s(EKE), s(NPP)	754.0	382.8	39.2%	30.3	2015.8	134075.5
dsm_nb_x_y_ms	Negative Binomial(0.116)	s(x), s(y), s(Depth), s(DistToCAS), s(SST), s(EKE), s(NPP)	752.6	383.0	40.73%	29.6	2015.8	137410.2

```
  summ <- summary(model)

  vp <- summary(dsm.var.prop(model, predgrid, off.set=predgrid$off.set))
  unconditional.cv.square <- vp$cv^2
  asymp.ci.c.term <- exp(1.96*sqrt(log(1+unconditional.cv.square)))
  asymp.tot <- c(vp$pred.est / asymp.ci.c.term,
                 vp$pred.est,
                 vp$pred.est * asymp.ci.c.term)

  data.frame(response = model$family$family,
             terms    = paste(rownames(summ$s.table), collapse=", "),
             AIC      = AIC(model),
             REML     = model$gcv.ubre,
             "Deviance_explained" = paste0(round(summ$dev.expl*100,2),"%"),
             "lower_CI" = round(asymp.tot[1],2),
             "Nhat" = round(asymp.tot[2],2),
             "upper_CI" = round(asymp.tot[3],2)
             )
}
```

```
# make a list of models (add more here!)
model_list <- list(dsm_nb_xy, dsm_nb_x_y, dsm_nb_xy_ms, dsm_nb_x_y_ms)
# give the list names for the models, so we can identify them later
names(model_list) <- c("dsm_nb_xy", "dsm_nb_x_y", "dsm_nb_xy_ms", "dsm_nb_x_y_ms")
per_model_var <- ldply(model_list, summarize_dsm_var, predgrid=predgrid_var)
```

```
kable(per_model_var, digits=1, booktabs=TRUE, escape=TRUE,
      caption = "Model performance: bivariate vs univariate spatial smooths without and with environ
  kable_styling(latex_options="scale_down")
```

10.5 Plotting

We can plot a map of the coefficient of variation, but we first need to estimate the variance per prediction cell, rather than over the whole area. This calculation takes a while!

```
# use the split function to make each row of the predictiond data.frame into
# an element of a list
predgrid_var_split <- split(predgrid_var, 1:nrow(predgrid_var))
var_split_nb_xy_ms <- dsm.var.prop(dsm_nb_xy_ms, predgrid_var_split,
```

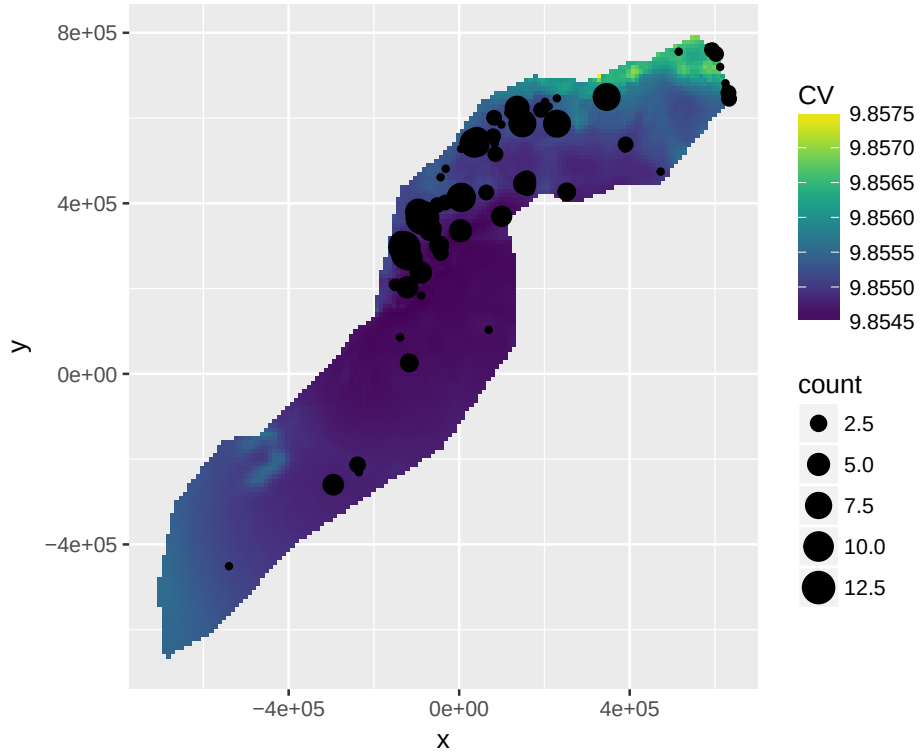


Figure 10.1: Uncertainty (CV) in prediction surface from bivariate spatial smooth with environmental covariates. Sightings overlaid.

```
                                    off.set=predgrid_var$off.set)
```

Now we have the per-cell coefficients of variation, we assign that to a column of the prediction grid data and plot it as usual:

```
predgrid_var_map <- predgrid_var
cv <- sqrt(var_split_nb_xy_ms$pred.var)/unlist(var_split_nb_xy_ms$pred)
predgrid_var_map$CV <- cv
p <- ggplot(predgrid_var_map) +
       geom_tile(aes(x=x, y=y, fill=CV, width=10*1000, height=10*1000)) +
       scale_fill_viridis() +
       coord_equal() +
       geom_point(aes(x=x,y=y, size=count),
                  data=dsm_nb_xy_ms$data[dsm_nb_xy_ms$data$count>0,])
```

```
## Warning: Ignoring unknown aesthetics: width, height
```

```
print(p)
```

Note that here we overplot the segments where sperm whales were observed (and scale the size of the point according to the number observed), using `geom_point()`.

We can also overplot the effort, which can be a useful way to see what the cause of uncertainty is. Though it may not only be caused by lack of effort but also covariate coverage, this can be useful to see.

First we need to load the segment data from the gdb

```
tracks <- readOGR("Analysis.gdb", "Segments")
```

```
## OGR data source with driver: OpenFileGDB
## Source: "Analysis.gdb", layer: "Segments"
## with 949 features
## It has 8 fields
```

```
tracks <- fortify(tracks)
```

We can then just add this to the plot object we have built so far (with +), but this looks a bit messy with the observations, so let's start from scratch:

```
p <- ggplot(predgrid_var_map) +
       geom_tile(aes(x=x, y=y, fill=CV, width=10*1000, height=10*1000)) +
       scale_fill_viridis() +
       coord_equal() +
       geom_path(aes(x=long, y=lat, group=group), data=tracks)
```

```
## Warning: Ignoring unknown aesthetics: width, height
```

```
print(p)
```

Try this with the other models you fitted and see what the differences are between the maps of coefficient of variation.

10.6 Save the uncertainty maps to raster files

As with the predictions, we'd like to save our uncertainty estimates to a raster layer so we can plot them in ArcGIS. Again, this involves a bit of messing about with the data format before we can save.

```
# setup the storage for the cvs
cv_raster <- raster(predictorStack)
# we removed the NA values to make the predictions and the raster needs them
# so make a vector of NAs, and insert the CV values...
cv_na <- rep(NA, nrow(predgrid))
cv_na[!is.na(predgrid$Depth)] <- cv
# put the values in, making sure they are numeric first
cv_raster <- setValues(cv_raster, cv_na)
# name the new, last, layer in the stack
names(cv_raster) <- "CV_nb_xy"
```

We can then save that object to disk as a raster file:

```
writeRaster(cv_raster, "cv_raster.img", datatype="FLT4S", overwrite=TRUE)
```

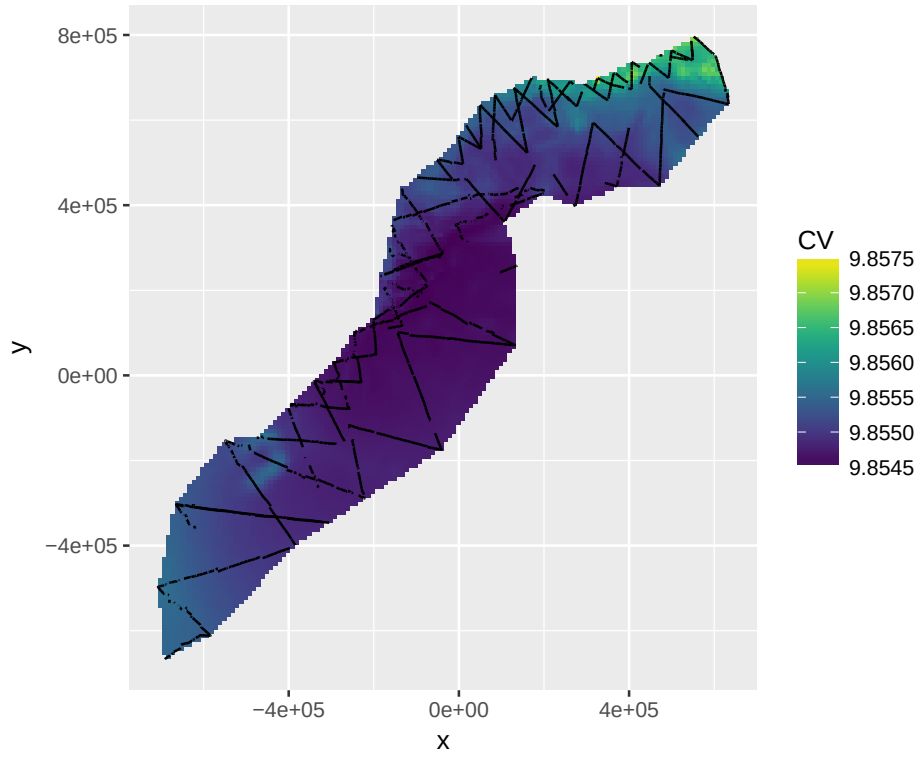


Figure 10.2: Uncertainty (CV) in prediction surface from bivariate spatial smooth with environmental covariates. Effort overlaid.

10.7 Extra credit

- `dsm.var.prop` and `dsm.var.gam` can accept arbitrary splits in the data, not just whole areas or cells. Make a `list` with two elements: one a `data.frame` of all the cells with $y > 0$ and one with $y \leq 0$. Estimate the variance for these regions. Note that you'll need to sum the offsets for each area to get the correct value to supply to `off.set=...`.

Chapter 11

Mark-recapture distance sampling of golftees

This document is designed to give you some pointers so that you can perform the Mark-Recapture Distance Sampling practical directly using the `mrds` package in R, rather than via the Distance graphical interface. I assume you have some knowledge of R, the `mrds` package, and Distance.

11.1 Golf tee survey

Luckily for us, the golf tee dataset is provided aspart of the mrds package, so we don't have to worry about obtaining the data from the Distance GolfteesExercise project.

Open R and load the mrds library and golf tee dataset.

```
library(mrds)
data(book.tee.data)
#investigate the structure of the dataset
str(book.tee.data)
```

```
List of 4
 $ book.tee.dataframe:'data.frame': 324 obs. of  7 variables:
  ..$ object  : num [1:324] 1 1 2 2 3 3 4 4 5 5 ...
  ..$ observer: Factor w/ 2 levels "1","2": 1 2 1 2 1 2 1 2 1 2 ...
  ..$ detected: num [1:324] 1 0 1 0 1 0 1 0 1 0 ...
  ..$ distance: num [1:324] 2.68 2.68 3.33 3.33 0.34 0.34 2.53 2.53 1.46 1.46 ...
  ..$ size    : num [1:324] 2 2 2 2 1 1 2 2 2 2 ...
  ..$ sex     : num [1:324] 1 1 1 1 0 0 1 1 1 1 ...
  ..$ exposure: num [1:324] 1 1 0 0 0 0 1 1 0 0 ...
 $ book.tee.region   :'data.frame': 2 obs. of  2 variables:
  ..$ Region.Label: Factor w/ 2 levels "1","2": 1 2
  ..$ Area        : num [1:2] 1040 640
 $ book.tee.samples  :'data.frame': 11 obs. of  3 variables:
  ..$ Sample.Label: num [1:11] 1 2 3 4 5 6 7 8 9 10 ...
```

```
  ..$ Region.Label: Factor w/ 2 levels "1","2": 1 1 1 1 1 1 2 2 2 2 ...
  ..$ Effort      : num [1:11] 10 30 30 27 21 12 23 23 15 12 ...
 $ book.tee.obs      :'data.frame': 162 obs. of  3 variables:
  ..$ object      : int [1:162] 1 2 3 21 22 23 24 59 60 61 ...
  ..$ Region.Label: int [1:162] 1 1 1 1 1 1 1 1 1 1 ...
  ..$ Sample.Label: int [1:162] 1 1 1 1 1 1 1 1 1 1 ...
```

```
#extract the list elements from the dataset into easy-to-use objects
detections <- book.tee.data$book.tee.dataframe
#make sure sex and exposure are factor variables
detections$sex <- as.factor(detections$sex)
detections$exposure <- as.factor(detections$exposure)
region <- book.tee.data$book.tee.region
samples <- book.tee.data$book.tee.samples
obs <- book.tee.data$book.tee.obs
```

We'll start by fitting the initial full independence model, with only distance as a covariate - just as was done in the "FI - MR dist" model in Distance. Indeed, if you did fit that model in Distance, you can look in the Log tab at the R code Distance generated, and compare it with the code we use here.

Feel free to use `?` to find out more about any of the functions used – e.g., `?ddf` will tell you more about the ddf function.

```
#Fit the model
fi.mr.dist <- ddf(method='trial.fi',mrmodel=~glm(link='logit',formula=~distance),
                  data=detections,meta.data=list(width=4))
#Create a set of tables summarizing the double observer data (this is what Distance does)
detection.tables <- det.tables(fi.mr.dist)
#Print these detection tables
detection.tables
```

```
Observer 1 detections
           Detected
            Missed Detected
  [0,0.4]        1       25
  (0.4,0.8]      2       16
  (0.8,1.2]      2       16
  (1.2,1.6]      6       22
  (1.6,2]        5        9
  (2,2.4]        2       10
  (2.4,2.8]      6       12
  (2.8,3.2]      6        9
  (3.2,3.6]      2        3
  (3.6,4]        6        2

Observer 2 detections
           Detected
            Missed Detected
  [0,0.4]        4       22
  (0.4,0.8]      1       17
```

```
  (0.8,1.2]      0       18
  (1.2,1.6]      2       26
  (1.6,2]        1       13
  (2,2.4]        2       10
  (2.4,2.8]      3       15
  (2.8,3.2]      4       11
  (3.2,3.6]      2        3
  (3.6,4]        1        7

Duplicate detections

  [0,0.4] (0.4,0.8] (0.8,1.2] (1.2,1.6]   (1.6,2]   (2,2.4] (2.4,2.8]
       21        15        16        20         8         8         9
(2.8,3.2] (3.2,3.6]   (3.6,4]
        5         1         1

Observer 1 detections of those seen by Observer 2
          Missed Detected Prop. detected
[0,0.4]        1       21      0.9545455
(0.4,0.8]      2       15      0.8823529
(0.8,1.2]      2       16      0.8888889
(1.2,1.6]      6       20      0.7692308
(1.6,2]        5        8      0.6153846
(2,2.4]        2        8      0.8000000
(2.4,2.8]      6        9      0.6000000
(2.8,3.2]      6        5      0.4545455
(3.2,3.6]      2        1      0.3333333
(3.6,4]        6        1      0.1428571
```

```
# They could also be plotted, but I've not done so in the interest of space
# plot(detection.tables)

#Produce a summary of the fitted detection function object
summary(fi.mr.dist)
```

```
Summary for trial.fi object
Number of observations                :  162
Number seen by primary                :  124
Number seen by secondary (trials)     :  142
Number seen by both (detected trials):  104
AIC                                   :  452.8094


Conditional detection function parameters:
             estimate        se
(Intercept)  2.900233 0.4876238
distance    -1.058677 0.2235722

                        Estimate          SE         CV
```

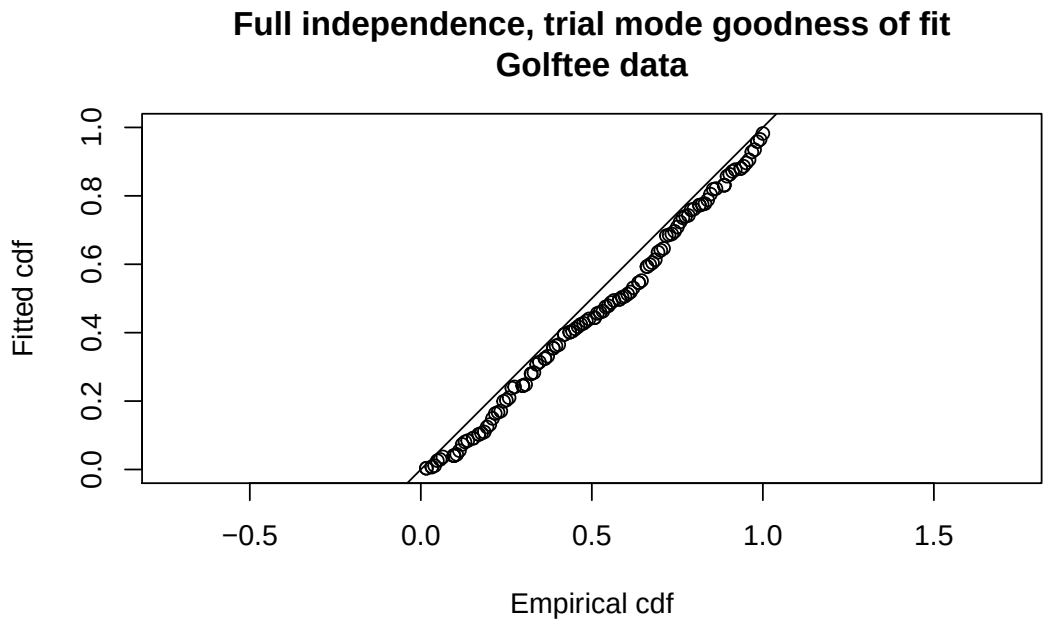


Figure 11.1: Goodness of fit (FI-trial) to golftee data.

```
Average p              0.6423252  0.04069409 0.06335434
Average primary p(0)   0.9478579  0.06109655 0.06445750
N in covered region  193.0486185 15.84826458 0.08209468
```

```
#Produce goodness of fit statistics and a qq plot
gof.result <- ddf.gof(fi.mr.dist,
                      main="Full independence, trial mode goodness of fit\nGolftee data")
```

```
chi.distance <- gof.result$chisquare$chi1$chisq
chi.markrecap <- gof.result$chisquare$chi2$chisq
chi.total <- gof.result$chisquare$pooled.chi
```

Abbreviated χ^2 goodness of fit assessment shows the χ^2 contribution from the distance sampling model to be 11.5 and the χ^2 contribution from the mark-recapture model to be 3.4. The combination of these elements produces a total χ^2 of 14.9 with 17 degrees of freedom, resulting in a P-value of 0.604

```
#Calculate density estimates using the dht function
tee.abund <- dht(fi.mr.dist,region,samples,obs)
kable(tee.abund$individuals$summary, digits=2,
      caption="Survey summary statistics for golftees")
```

```
kable(tee.abund$individuals$N, digits=2,
      caption="Abundance estimates for golftee population with two strata")
```

Now, see if you can work out how to change the call to ddf to fit the other models mentioned in the

Table 11.1: Survey summary statistics for golftees

Region	Area	CoveredArea	Effort	n	ER	se.ER	cv.ER	mean.size	se.mean
1	1040	1040	130	229	1.76	0.12	0.07	3.18	0.21
2	640	640	80	152	1.90	0.33	0.18	2.92	0.23
Total	1680	1680	210	381	1.81	0.14	0.08	3.07	0.15

Table 11.2: Abundance estimates for golftee population with two strata

Label	Estimate	se	cv	lcl	ucl	df
1	356.52	32.35	0.09	294.54	431.53	17.13
2	236.64	44.14	0.19	147.33	380.09	5.06
Total	593.16	60.38	0.10	478.32	735.57	16.06

exercise, and then write code to enable you to compare the models and select among them.

11.2 Crabeater seal survey

This analysis is described in Borchers et al. (2005) Biometrics paper of aerial survey data looking for seals in the Antarctic pack ice. There were four observers in the plane, two on each side (front and back).

The data from the survey has been saved in a `.csv` file. This file can be easily read into R, and with the `checkdata()` function, the information to construct the region, sample, and observation table can be extracted. Note that these tables are only needed when estimating abundance by scaling up from the covered region to the study area.

```
library(Distance)
crabseal <- read.csv("crabbieMRDS.csv")
#  Half normal detection function, 700m truncation distance,
#      logit function for mark-recapture component
crab.ddf.io <- ddf(method="io", dsmodel=~cds(key="hn"),
                   mrmodel=~glm(link="logit", formula=~distance),
                   data=crabseal, meta.data=list(width=700))
summary(crab.ddf.io)
```

```
Summary for io.fi object
Number of observations   :  1740
Number seen by primary   :  1394
Number seen by secondary :  1471
Number seen by both      :  1125
AIC                      :  3011.463


Conditional detection function parameters:
               estimate          se
```

```
(Intercept)  2.107762345 0.0994391200
distance    -0.003087713 0.0003159216

                        Estimate          SE          CV
Average primary p(0)   0.8916554 0.009606428 0.010773701
Average secondary p(0) 0.8916554 0.009606428 0.010773701
Average combined p(0)  0.9882614 0.002081614 0.002106339


Summary for ds object
Number of observations :  1740
Distance range         :  0  -  700
AIC                    :  22314.4

Detection function:
 Half-normal key function

Detection function parameters
Scale coefficient(s):
            estimate        se
(Intercept) 5.828703 0.0268578

           Estimate         SE         CV
Average p 0.5845871 0.01247837 0.02134562


Summary for io object
Total AIC value :  25325.86

                         Estimate          SE         CV
Average p               0.5777249  0.01239179 0.02144929
N in covered region 3011.8139211 79.84197966 0.02650960
```

Goodness of fit could be examined in the same manner as the golf tees by the use of `ddf.gof(crab.ddf.io)` but I have not shown this step.

Following model criticism and selection, estimation of abundance ensues. the estimates of abundance for the study area are arbitrary because inference of the study was restricted to the covered region. Hence the estimates of abundance here are artificial, but if we wished to produce them, we would need to produce the region, sample, and observation tables and apply Horvitz-Thompson like estimators to produce estimates of $\hat{N}$. The use of `covert.units` adjusts the units of perpendicular distance measurement (m) to units of transect effort (km). Be sure to perform the conversion correctly or your abundance estimates will be off by orders of magnitude.

```
tables <- Distance:::checkdata(crabseal[crabseal$observer==1,])
crab.ddf.io.abund <- dht(region=tables$region.table,
                         sample=tables$sample.table, obs=tables$obs.table,
                         model=crab.ddf.io, se=TRUE, options=list(convert.units=0.001))
kable(crab.ddf.io.abund$individuals$summary, digits=3,
      caption="Summary information from crabeater seal aerial survey.")
```

Table 11.3: Summary information from crabeater seal aerial survey.

Region	Area	CoveredArea	Effort	n	ER	se.ER	cv.ER	mean.size	se.mean
1	1e+06	8594.082	6138.63	2053	0.334	0.033	0.097	1.18	0.013

Table 11.4: Crabeater seal abundance estimates for study area of arbitrary size.

Label	Estimate	se	cv	lcl	ucl	df
Total	413493.2	41201.49	0.09964248	339670.9	503359.6	128.6257

```
kable(crab.ddf.io.abund$individual$N, digits=3,
      caption="Crabeater seal abundance estimates for study area of arbitrary size.")
```

References

Amodio, Sonia, Massimo Aria, and Antonio D'Ambrosio. 2014. "On Concurvity in Nonlinear and Nonparametric Regression Models." *Statistica* 74 (1): 85–98. doi:10.6092/issn.1973-2201/4599.

Borchers, D. L., J. L. Laake, C. Southwell, and C. G. M. Paxton. 2005. "Accommodating Unmodeled Heterogeneity in Double-Observer Distance Sampling Surveys." *Biometrics* 62 (2). Wiley-Blackwell: 372–78. doi:10.1111/j.1541-0420.2005.00493.x.

Faraway, J. J. 2006. *Extending the Linear Model with R*. Chapman & Hall / CRC. http://www.maths.bath.ac.uk/%7Ejjf23/ELM/.

Palka, Debra L. 2006. "Summer Abundance Estimates of Cetaceans in the Us North Atlantic Operating Areas." Research report. US Dept of Commerce, Northeast Fisheries Science Center.

Wood, Simon N. 2006. *Generalized Additive Models: An Introduction with R*. CRC/Chapman & Hall.

Zuur, Alain F., Elena N. Ieno, Neil Walker, Anatoly A. Saveliev, and Graham M. Smith. 2009. *Mixed Effects Models and Extensions in Ecology with R*. Springer.