An Introduction to R: Examples for Actuaries

Nigel De Silva

28 Jan 2006
version 0.1

Contents

1 Introduction

This is a very brief introduction to R via a series of examples which are relevant to actuarial work. These examples are aimed at R "newbies" so:

- only a small fraction of R's capabilities are examined in these examples.
- there are many ways of achieving the same objective in R. These examples often only consider a single approach and for educational purposes may not be the most elegant solutions.

This document is no substitute to the many excellent resources available online and it is strongly recommended that you review these. For example, a much more comprehensive introduction to R is provided in "Using R for Data Analysis and Graphics – Introduction, Code and Commentary" by J H Maindonald (http://cran.r-project.org/doc/contrib/usingR.pdf).

Hyperlinks are provided throughout this series to useful online resources (sometimes with the web address for those reviewing paper copies). Further help is widely available online.

1.1 What is R?

R is an integrated suite of software facilities for data manipulation, calculation and graphical display. Among other things it has:

- an effective data handling and storage facility,
- a suite of operators for calculations on arrays, in particular matrices,
- a large, coherent, integrated collection of intermediate tools for data analysis,
- graphical facilities for data analysis and display either on-screen or on hardcopy, and
- a well-developed, simple and effective programming language which includes conditionals, loops, user-defined recursive functions and input and output facilities.

R can be regarded as an implementation of the S language which was developed at Bell Laboratories and forms the basis of the S-Plus systems.

R is available, from http://www.r-project.org/, as Free Software under the terms of the Free Software Foundation's GNU General Public License. It runs on a wide variety of UNIX platforms and similar systems (including FreeBSD and Linux), Windows and MacOS.

1.2 R Packages

So far we have not mentioned statistics, yet many people use R as a statistics system. R's developers prefer to think of it of an environment within which many classical and modern statistical techniques have been implemented. A few of these are built into the base R environment, but many are supplied as *packages*.

All R functions and datasets are stored in packages. For instance, in section 3.4, we use a package called "evir", which contains a number of useful functions for conducting extreme value analysis. Only when a package is loaded are its contents available. This greatly improves R's efficiency as it doesn't have to store the many thousands of potential commands available in the hundreds of available packages.

There are about 25 packages supplied with R (called "standard" and "recommended" packages) and many more are available through the CRAN family of Internet sites (via http://CRAN.R-project.org) and elsewhere.

Packages can be installed and loaded via the *Package* menu item, or though R commands.

1.3 Online Resources

The R project homepage (http://www.r-project.org/) contains contributed documents and manuals from various authors.

The GIRO Toolkit Working Party has an R toolkit page of its wiki, that discusses some online R resources. In particular:

- a Google search including the term "R", usually results in some useful results.

- The social book marking website Del.icio.us search produces more useful general links:
  a) The most popular R sites bookmarked on delicious
  b) All R tags on delicious (good for seeing what's just been tagged)

2 Some Preliminaries

2.1 Starting R in Windows

Please refer to the R Project website at http://www.R-project.org/ for instructions on installing R and any add-on packages.

There are several ways to input commands to R

- the *command window*. This is available immediately after starting R and it contains the command line prompt ">", which is an invitation to start typing. For example, if we enter:

```
> 2 +       2
[1] 4
>
```

  we obtain the answer 4! The `[1]` indicates that the first element of the response follows. The last ">" indicates that R is ready for another command.

- The *script window*. This is a simple text editor with which to write and save more complex commands of series of commands. It can be accessed from the *File* menu item. You can create a new script or open an existing one. In this window, we can type a number of commands or an entire program.

  To execute the commands, highlight them and click on the "*Run line or selection*" icon in the middle of the script file editor toolbar. The commands are automatically copied over to the command window and run.

2.2 Getting help with functions and features

To get information on any specific named function, for example *plot*, the command is

```
> help(plot)
```

An alternative is

```
> ?plot
```

For a feature specified by special characters, the argument must be enclosed in double or single quotes, making it a “character string”: This is also necessary for a few words with syntactic meaning including if, for and function.

```
> help("[[")
```

Help is available in HTML format by running or via the

```
> help.start()
```

which will launch a Web browser that allows the help pages to be browsed with hyperlinks. The `Search Engine and Keywords' link in the page loaded by `help.start` is particularly useful as it is contains a high-level concept list which searches though available functions. It can be a great way to get your bearings quickly and to understand the breadth of what R has to offer.

The examples on a help topic can normally be run by

```
> example(topic)
```

This series will not discuss all functions used in the examples. Neither will it discuss all of the options available for each function. Use the help functions, detailed above, to get further information as necessary.

2.3 R Data Structures

2.3.1 Data Classes

As an object orientated language, everything in R is an object. Each object has a class.

The simplest data objects are one-dimensional arrays called *vectors*, consisting of any number of *elements*. For example, the calculation:

```
> 2 + 2
[1] 4
```

results in a vector, from the *numeric* class (as it contains a number), with just one element. Note that the command "`2+2`" is itself and object of the *expression* class

The simplest elements produce vectors of the following classes:

- *logical*: The values T (or TRUE) and F (or FALSE).
- *integer*: Integer values such as 3 or -4.
- *numeric*: Floating-point real numbers (double-precision by default). Numerical values can be written as whole numbers (for example, 3., -4.), decimal fractions (4.52, -6.003), or in scientific notation (6.02e23, 8e-47).
- *complex*: Complex numbers of the form a + bi, where a and b are integers or numeric (for example, 3 + 1.23i).
- *character*: character strings enclosed by matching double quotes (") or apostrophes (’), for example, "Alabama", ’idea’.

Two other elements which are particularly useful are:

- *factors*: These represent labelled observations. For example sex is a factor, generally incorporating two levels: male and female. These are generally used to represent qualitative effects in models.
- *ordered factors*: A factor where the levels are ordered. For example there may be three responses to a question about quality, high, medium or low, but each level is not necessarily on a linear scale.

2.3.2 Vectors

The simplest type of data object in R is a vector, which is simply an ordered set of values. Some further examples of creating vectors are shown below:

```
> 11:20
[1] 11 12 13 14 15 16 17 18 19 20
```

This creates a numeric vector containing the elements 11 to 20. The ":" is a shorthand for the explicit command, `seq(from=11, to=20, by=1)`. Vectors can be assigned a name (case sensitive) via the assignment operator ("`<-`"), for example:

```
> x <- 11:20
> y <- c(54, 16, 23, 34, 87)  # "c" means "combine"
> z <- c("apple", "bear", "candle")
```

Note: The "#" can be used to make comments in your code. R ignores anything after it on the same line.

To display a vector, use its name. To extract subsets of vectors, use their numerical indices with the subscript operator "[" as in the following examples.

```
> z
[1] "apple"  "bear"   "candle"
> x[4]
[1] 14
> y[c(1,3,5)]
[1] 54 23 87
```

The number of elements and their *mode* completely define the data object as a vector. The class of any vector is the mode of its elements:

```
> class(c(T,T,F,T))
[1] "logical"
> class(y)
[1] "numeric"
```

The number of elements in a vector is called the length of the vector and can be obtained for any vector using the length function:

```
> length(x)
[1] 10
```

Vectors may have named elements.

```
> temp <- c(11, 12, 17)
> names(temp) <- c("London", "Madrid", "New York")
> temp
  London   Madrid New York
      11       12       17
```

Operations can be performed on the entire vector as a whole without looping through each element. This is important for writing efficient code as we will see later. For example, a conversion to Fahrenheit can be achieved by:

```
> 9/5 * temp + 32
  London   Madrid New York
    51.8     53.6     62.6
```

2.3.3 Matrices

An extension of the vector is the *matrix* class. We can create a matrix from the vector `x` as shown below:

```
> matrix(x, nrow=5)
     [,1] [,2]
[1,]   11   16
[2,]   12   17
[3,]   13   18
[4,]   14   19
[5,]   15   20
```

Or alternatively, by:

```
> dim(x) <- c(5,2)
> x
     [,1] [,2]
[1,]   11   16
[2,]   12   17
[3,]   13   18
[4,]   14   19
[5,]   15   20
```

We can join matrices via `cbind` or `rbind`. As for vectors, we can extract an element using subscripts, or perform operations on all elements:

```
> x[3,2]
[1] 18
> x[3, ]    # Omitting the column index prints the entire row
[1] 13 18
> x-10
     [,1] [,2]
[1,]    1    6
[2,]    2    7
[3,]    3    8
[4,]    4    9
[5,]    5   10
```

Alternatively, matrix operations are possible, for example:

```
> t(x)      # Transpose
     [,1] [,2] [,3] [,4] [,5]
[1,]   11   12   13   14   15
[2,]   16   17   18   19   20
> t(x) %*% x      # Multiplication
     [,1] [,2]
[1,]  855 1180
[2,] 1180 1630
```

Many more operations are possible, for example solving linear equations, eigenvalues and eigenvectors, decompositions, etc.

2.3.4 Arrays

Arrays are a further abstraction of matrices and can be created in similar ways. For example:

```
> array(1:12, dim=c(2,3,2))
, , 1

     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

, , 2

     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12
```

As with matrices, various operations are possible for arrays.

2.3.5 Data Frames

Data frames are of fundamental importance when it comes to most modelling and graphical functions in R. Until now all data strictures have been atomic in that they contain data of just one mode, such as integers or characters. Data frames are two dimensional tables in which each column can take on different modes.

We can create a data frame as follows:

```
> Temps <- data.frame(town = c("London", "Madrid", "New York"),
+ temp = c(11, 12, 17))
> Temps
      town temp
1   London   11
2   Madrid   12
3 New York   17
```

Note that the command is split over two lines. As the final closing parentheses for the `data.frame` function was not provided in the first command, R expected further instructions, in this case details for the second column of temperatures.

The columns are named town and temp respectively. We can extract subsets of this data frame via subscripts (as for matrices):

```
> Temps[1,2]
[1] 11
> Temps[2,1]
[1] Madrid
Levels: London Madrid New York
```

Notice that the “town” column is not actually a vector of characters, but a vector of factors with three levels, listed below the result.

An alternative method of extraction is to consider that each column of the data frame is a vector with a name. It can be accessed via the "$" operator, and the result treated as a vector.

```
> Temps$temp
[1] 11 12 17
> Temps$temp[2]
[1] 12
```

Often packages come with datasets. For example the MASS library contains a dataset called Insurance. We can access the first five rows of this dataset via:

```
> library(MASS)
> Insurance[1:5,]
  District  Group   Age Holders Claims
1        1    <1l   <25     197     38
2        1    <1l 25-29     264     35
3        1    <1l 30-35     246     20
4        1    <1l   >35    1680    156
5        1 1-1.5l   <25     284     63
```

This is a dataset of motor claims where, District is a factor, Group and Age are ordered factors and the last two columns are integers.

```
> class(Insurance$Age)
[1] "ordered" "factor"
> levels(Insurance$Age)
[1] "<25"   "25-29" "30-35" ">35"
```

2.3.6 Lists

Lists make it possible to collect an arbitrary set of R objects together under a single name. You might for example collect together vectors of several different modes and lengths, scalars, matrices or more general arrays, functions, etc. Lists can be, and often are, a rag-tag of different objects.

As an illustration the list object that R creates as output from the `attributes` function.

```
> attributes(Temps)
$names
[1] "town" "temp"

$row.names
[1] "1" "2" "3"

$class
[1] "data.frame"
```

In this case, the elements are all vectors. We can access a list's elements via subscripts, this time using the "[[" operator, or via names, using the "$" operator. For example:

```
> attributes(Temps)[[1]]
[1] "town" "temp"
> attributes(Temps)$row.names
[1] "1" "2" "3"
```

2.3.7 Other Classes

There are many other object classes that you will come across in R. As in any other object orientated programming language, classes can be defined by the user and contributed packages typically create special classes used by their functions.

Generally, you may never need to worry about classes. User defined classes are typically lists. The elements can be identified via the `names` function and then accessed as described above.

```
> names(attributes(Temps))
[1] "names"     "row.names" "class"
```

2.4 R Graphics

R has an amazing array of graphical capabilities and is widely used to produce graphics for publication. As a simple introduction to R's capabilities, try the following commands for a demo.

```
> demo(graphics)
> demo(persp)
```

These open a graphics window. Clicking on the window or pressing enter will allow you the view the various demo graphs. A couple of online resources for R graphs are listed below:

- R Graph Gallery (http://addictedtor.free.fr/graphiques/index.php)
- R Graphics by Paul Murrell (http://www.stat.auckland.ac.nz/~paul/RGraphics/rgraphics.html)

This series of examples will introduce a few of the simpler graphical functions. A more comprehensive introduction is available in the recommended reading, Section 1.1.

2.5 Getting data into R

There are various ways of getting data into R. For small datasets, the command line might be the most efficient method:

```
> x <- c(7.82,8.00,7.95)
> x
[1] 7.82 8.00 7.95
> x <- scan()      # A blank line indicates the end of the data
1: 7.82
2: 8.00
3: 7.95
4:
Read 3 items
> x
[1] 7.82 8.00 7.95
```

For larger datasets, it is better to import the data from a file. R can read data stored in text (ASCII) files. It can also read files in other formats (Excel, SAS, SPSS, etc), and access SQL-type databases, but the functions needed for this are not in the package base. These functionalities are very useful for a more advanced use of R, but are beyond the scope of this introduction.

Some useful functions for achieving for reading data are `scan` and `read.table`. There are some standard variations of the `read.table` function which are summarised in the help file.

I generally use the `read.csv` variant. *Comma separated values (or csv)* files can be created from any spreadsheet or database application. An example of this is included below.

> **Note**: A character preceded by a "\" is considered a special character in R. For example, "\n" designates a new line, "\t" designates a tab, etc. In order to interpret a slash in a string correctly, it has to be preceded by another slash. Therefore, the correct path string requires the "\\" seen in the command above.

```
> weld <- read.csv("C:\\R\\weld.csv")
> weld
      x   y
1  7.82 3.4
2  8.00 3.5
3  7.95 3.3
4  8.07 3.9
5  8.08 3.9
6  8.01 4.1
7  8.33 4.6
8  8.34 4.3
9  8.32 4.5
10 8.64 4.9
11 8.61 4.9
12 8.57 5.1
13 9.01 5.5
14 8.97 5.5
```

```
15 9.05 5.6
16 9.23 5.9
17 9.24 5.8
18 9.24 6.1
19 9.61 6.3
20 9.60 6.4
21 9.61 6.2
> class(weld)
[1] "data.frame"
```

The data has been converted to a data frame. The rows have been automatically named by their number. However a descriptive names could be used instead.

R tries to determine the class of each data column. Sometimes this requires fine tuning via the `colClasses` argument to `read.csv`. Further details are available in the help file. Alternatively, conversions can be done afterwards, via functions such as `as.character`.

Add-on packages (e.g. *foreign* or *gdata*) allow the user to read data from various file formats native to other packages, such as Excel, SAS, Stata, etc.

2.6 Getting data out of R

2.6.1 Text Output

There are various ways to achieve this. For example, you can divert output from the screen to a file via the `sink` function. However, an easier way to do this is via the `write.table` function, or its variant `write.csv`.

This example outputs the weld data frame to a csv file:

```
> write.csv(weld, "C:\\R\\weld 2.csv")
```

2.6.2 Graphical Output

Again there are various ways to achieve this. The simplest method seems to be to write right-click on the graph window and choose one of the options to copy or save the picture.

For example, we can create a plot of the weld data by:

```
> plot(weld$x, weld$y, main="Scatterplot of weld data")
```

2.7 Examples Series

The following series of examples will provide simple introductions into possible uses for R. Functions will be introduced with little commentary regarding the syntax, so don't forget to use the help options.

Commands will be taken from the script editor window rather than the command line window, as in this section. As they will not contain the ">" prompt, you can copy the commands directly from this text into R and run the examples yourself.

Some examples rely on data files supplied with this text. The code assumes that you have these files in your "C:\R" directory. Adjust the code as necessary to reflect the actual location of these files (see section 2.5 for details).

3 Examples

3.1 R as a Calculator

In Section 2, there were several examples of using R as a calculator. A particularly useful feature is that operations can be performed on vectors as a whole.

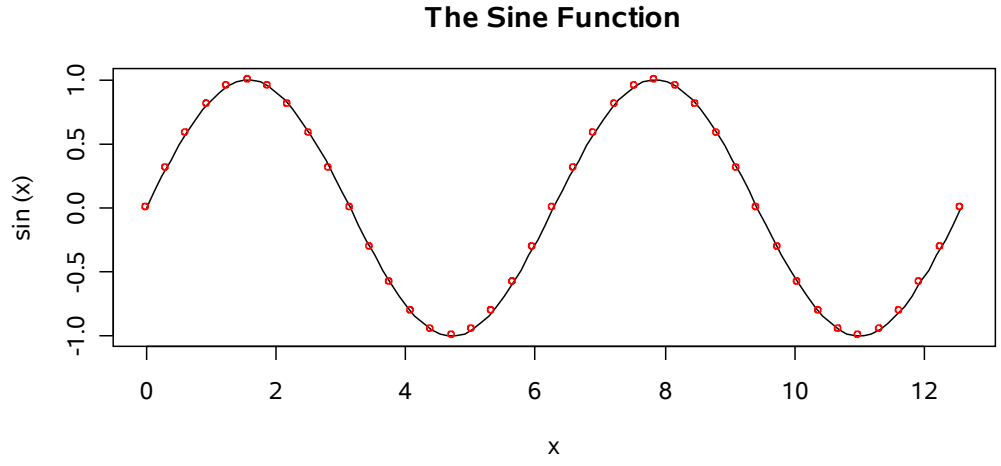
For example consider the *sine* function. We can calculate the values of this function for various values of x from 0 to 4π:

```
x <- pi*seq(0,4, by=0.1)
sin(x)

 [1]  0.000000e+00  3.090170e-01  5.877853e-01
 [4]  8.090170e-01  9.510565e-01  1.000000e+00
 [7]  9.510565e-01  8.090170e-01  5.877853e-01
[10]  3.090170e-01  1.224606e-16 -3.090170e-01
[13] -5.877853e-01 -8.090170e-01 -9.510565e-01
[16] -1.000000e+00 -9.510565e-01 -8.090170e-01
[19] -5.877853e-01 -3.090170e-01 -2.449213e-16
[22]  3.090170e-01  5.877853e-01  8.090170e-01
[25]  9.510565e-01  1.000000e+00  9.510565e-01
[28]  8.090170e-01  5.877853e-01  3.090170e-01
[31]  3.673819e-16 -3.090170e-01 -5.877853e-01
[34] -8.090170e-01 -9.510565e-01 -1.000000e+00
[37] -9.510565e-01 -8.090170e-01 -5.877853e-01
[40] -3.090170e-01 -4.898425e-16
```

The `seq` function, returns a vector representing the sequence from 0 to 4 in increments of 0.1. Assigning the points to the variable x was an unnecessary step, included for clarity. The command could have been performed in one line.

We can plot a graph of the sine function, the cosine function and plot our calculated points on it via the following commands:

```
plot(sin, xlim=c(0,4*pi), main="The Sine Function")
points(x, sin(x), col="red")
```



Try finding a root of this equation using the `uniroot` function.

3.2 Data manipulation

This simple example considers a dataset called "mammals.csv". It contains details of body weight (kg) and brain weight (g) for a variety of mammals. First we load up the dataset and consider its size.

```
mammals <- read.csv("C:\\R\\mammals.csv")
dim(mammals)

[1] 65  2
```

Then we view the first five rows, because for large datasets it will not be feasible to look at the entire table.

```
mammals[1:5, ]# View the first five rows

                   body brain
Artic fox         3.385  44.5
Owl monkey        0.480  15.5
Mountian beaver   1.350   8.1
Cow             465.000 423.0
Grey wolf        36.330 119.5
```

We can then create a quick summary of the data:

```
summary(mammals)

      body              brain
 Min.   :   0.005   Min.   :   0.14
 1st Qu.:   0.750   1st Qu.:   4.50
 Median :   3.500   Median :  17.50
 Mean   : 193.969   Mean   : 326.26
 3rd Qu.:  55.500   3rd Qu.: 172.00
 Max.   :6654.000   Max.   :5712.00
                    NA's   :   2.00
```

The resulting output provides sample statistics of the numerical columns of data. The output for each column depends on the type of data held.

Notice that the brain column contains "NA's". In R, a missing value is denoted by NA. To identify which data points have NA's we can use the following code.

```
mammals[is.na(mammals$brain), ]

          body brain
Lion       170    NA
Sea Otter   43    NA
```

As mentioned in section 2.3.5, we can extract parts of a data frame via its subscripts. However we don't have to provide numerical subscripts. We can provide a logical vector (i.e. one containing T or F) and only the T's are shown.

The function `is.na(mammals$brain)`, tests each element of the `mammals$brain` vector to see whether or not it is an NA element. It returns a logical vector showing the result of this comparison.

Note: Generally the "==" operator is used to test for equality. However, all arithmetical operations on an NA value will result in an NA output. To get a valid logical output we therefore must use the `is.na` function

We can remove the missing data points, and confirm the size of the data frame to check that the correct number of rows are remaining.

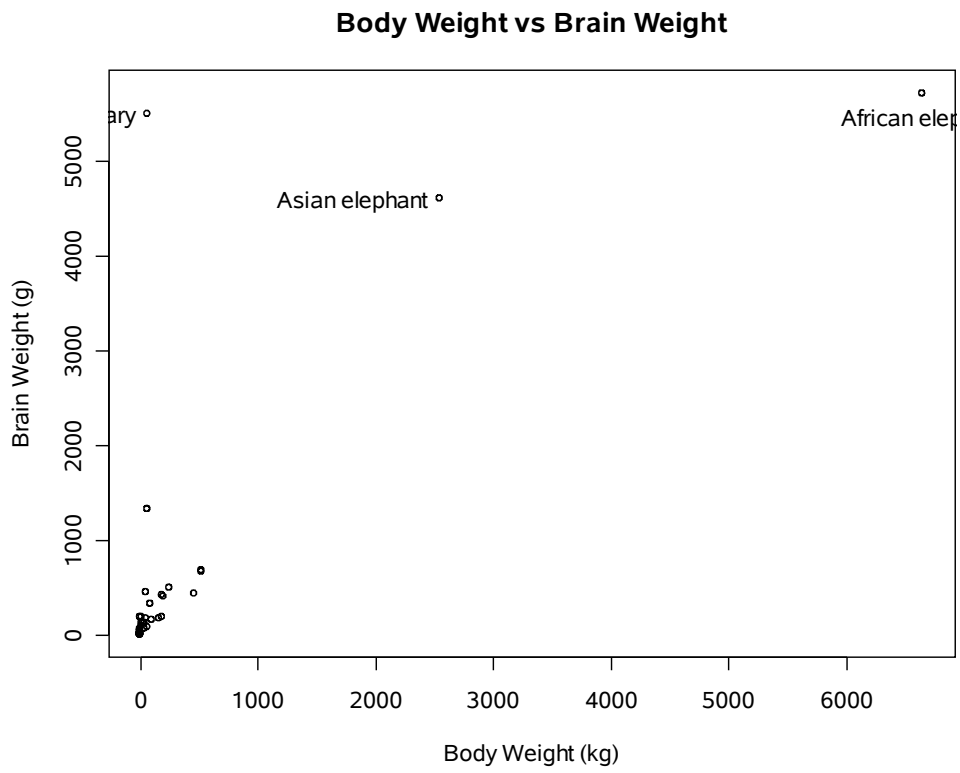
```
mammals <- mammals[!is.na(mammals$brain), ] # ! means NOT
dim(mammals)

[1] 63  2
```

The syntax to the right of the assignment operator selects the non-NA rows of the data. By assigning the result back to "mammals", we have overwritten the original table resulting in a table with only 63 rows (rather than 65).

It might be nice to plot a graph of the data and identify any outliers.

However as we're only dealing with a single data frame, and it is tedious typing "`mammals$`" before each vector we are interested in, we can *attach* the data frame to R's search path. This makes a copy of the dataframe in the memory and makes all vectors available directly.

```
attach(mammals)
plot(body, brain, main="Body Weight vs Brain Weight",
  xlab="Body Weight (kg)", ylab="Brain Weight (g)")
identify(body, brain, rownames(mammals))
# Left click on data points to identify, right click to stop

[1]  1  5 25
```



It looks as though the two right outliers are reasonable, belonging to elephants. However, the name of the remaining outlier is obscured. The `identify` function also returned a vector of numbers, corresponding to the rows of the data frame containing the selected points. Let us examine these:

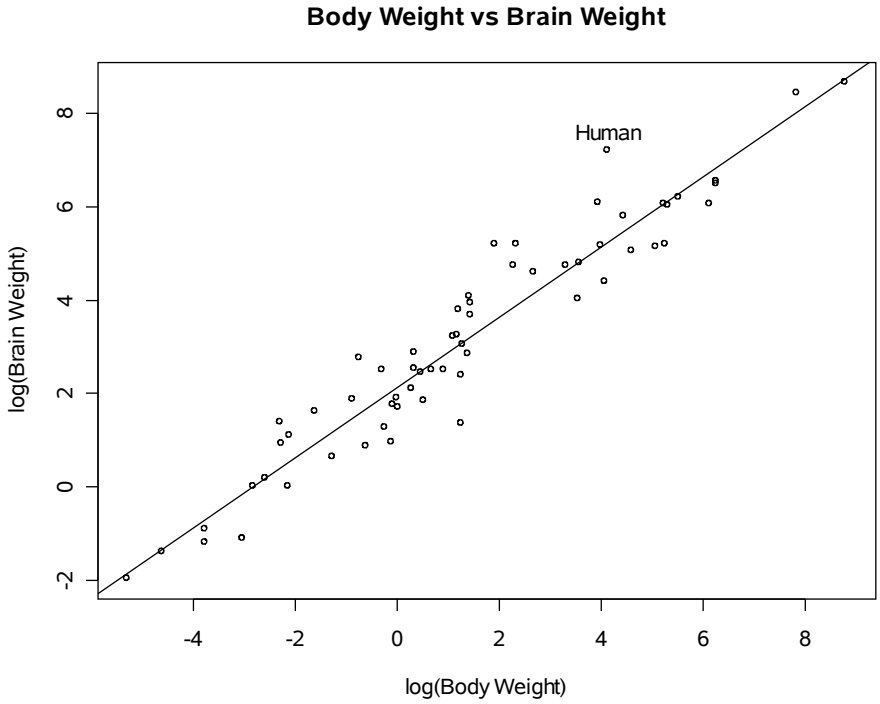
```
mammals[c(1,5,25), ]
```

```
                        body brain
African elephant        6654  5712
Asian elephant          2547  4603
Greater Spotted Actuary   70  5500
```

It appears that the Greater Spotted Actuary has a very large brain! Or perhaps there is an error. We remove this data point and consider plotting the logarithm of the variables. Conveniently, the data appears to follow a straight line, so perhaps we could fit a linear regression model?

```
# -'ve subscript means "except" row
mammals <- mammals[-25, ]

# Create vector of logs
mammals$lbody <- log(mammals$body)
mammals$lbrain <- log(mammals$brain)

# Reattach data frame as now altered
attach(mammals)

plot(lbody, lbrain, main="Body Weight vs Brain Weight",
  xlab="log(Body Weight)", ylab="log(Brain Weight)")
identify(lbody, lbrain, rownames(mammals))
```

```
mammals.lm <- lm(lbrain ~ lbody)
abline(mammals.lm)
```



3.3 Examining Distributions

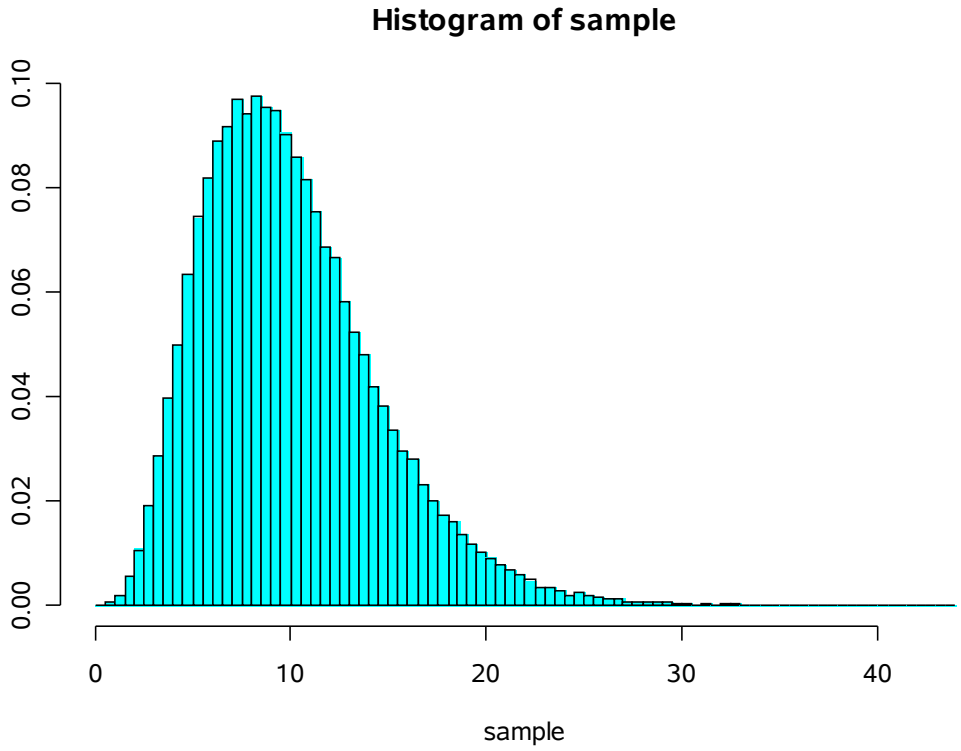
A common requirement is for actuaries to fit a probability distribution to some loss data. This section provides some simple examples of how we might achieve this in R.

First we need some data. For this example we shall create a random sample (n=100,000) of data from a Gamma distribution.

```
n <- 100000
sample <- rgamma(n, rate=0.5, shape=5)
```

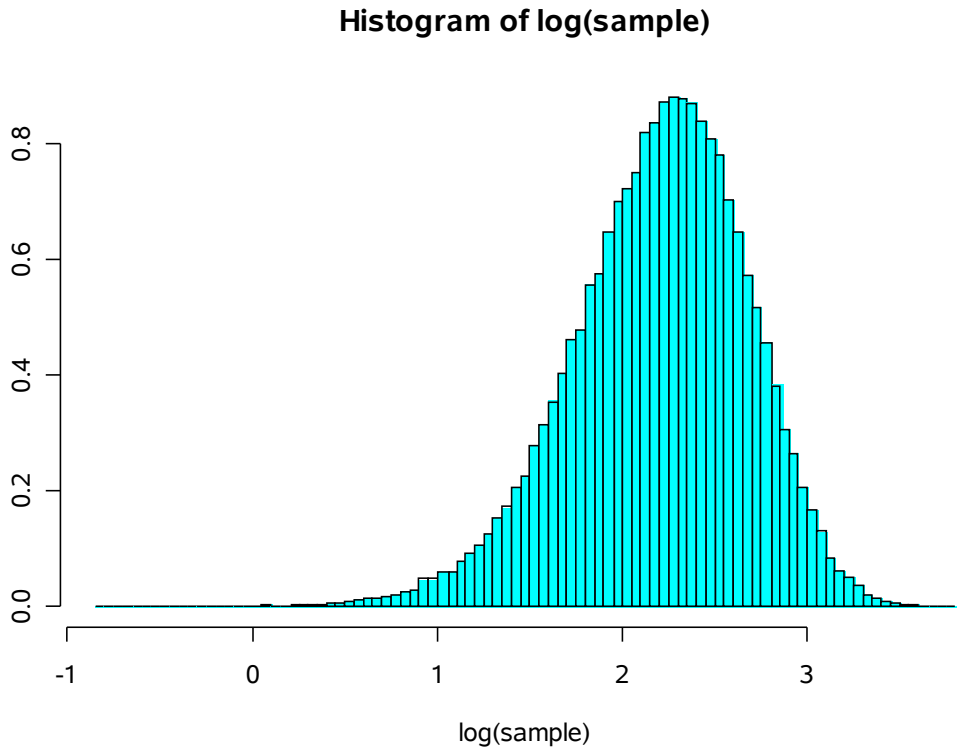
We will then examine this distribution via a histogram. We could use the standard function `hist`, to create the graph, but the rule that this function uses to produce "nice" cut-points between the bins often produces too few bins. Whilst this is adjustable via the breaks parameter, I find the function `truehist` in the MASS package produces better default results. By default the y-axis shows the density of observations.

```
library(MASS)
truehist(sample, main="Histogram of sample")
```



Observing that the data has a heavy tail, we might consider a simple transformation to normalise the data, such as taking logarithms. If the data comes from a lognormal population, we would expect a nice bell curve.

```
truehist(log(sample), main="Histogram of log(sample)")
```



The log transform doesn't appear to work well here. An alternative is to consider a power transformation of the form x^n, known as a Box-Cox transformation. We can find a suitable power via the command: `boxcox(sample ~ 1)`.

Instead we shall try fitting some other heavy tailed distributions such as the Weibull, Gamma. We could fit distributions manually via method of moments or use maximum likelihood estimation.

Oddly, R doesn't have standard functions for fitting distributions. However methods are available in various add on packages.

- `mle` – in the stats4 package. Requires the user to define the log-likelihood function manually.
- `fitdistr` – in the MASS package.

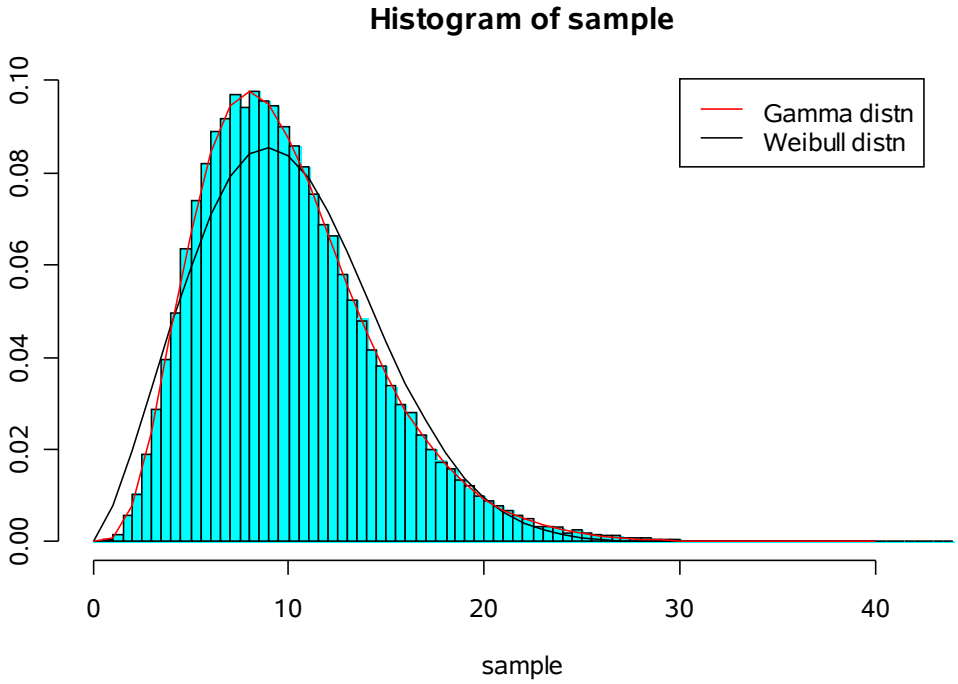
As the `fitdistr` function requires less information from the user, we shall use this one. For example:

```
fitdistr(sample, "Weibull")
      shape           scale
   2.360351150   11.277036784
 ( 0.005528084) ( 0.015984577)
```

The output is an object of the "fitdistr" class. This is essentially a list containing two vectors, "estimate" "sd". We can plot density curves for the fitted distribution via the following code:

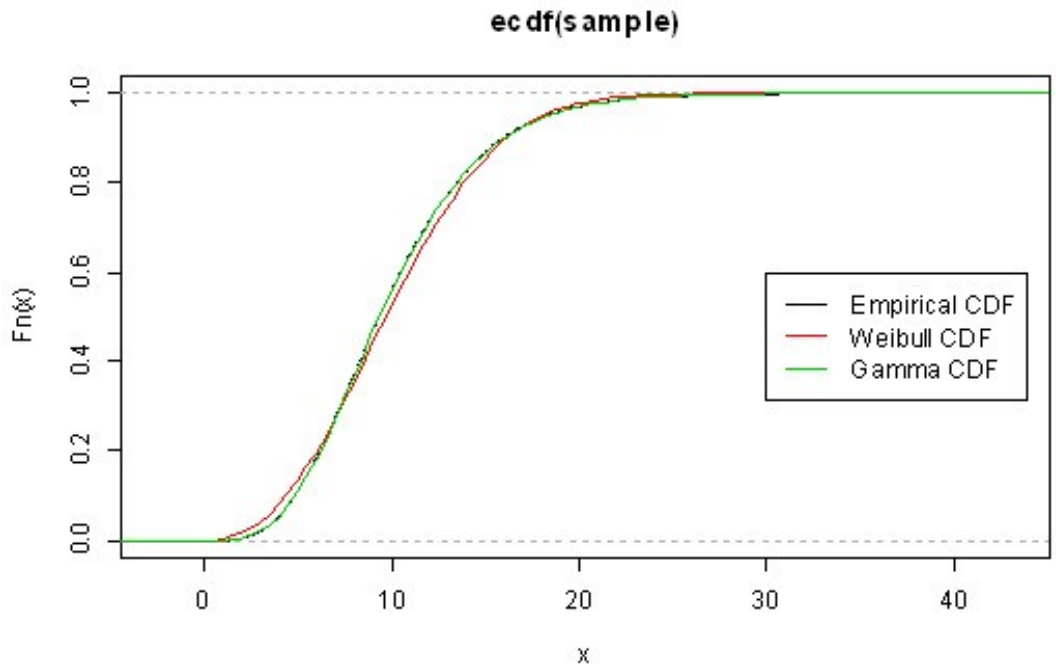
```
# Fit distributions
```

```
fit.w <- fitdistr(sample, "Weibull")
fit.g <- fitdistr(sample, "gamma")

# Caclulate points for density curves
x <- seq(from=0, to=40, by=1)
y.w <- dweibull(x, shape=fit.w$estimate[1],
scale=fit.w$estimate[2])
y.g <- dgamma(x, shape=fit.g$estimate[1],
rate=fit.g$estimate[2])

# Draw histogram and density curves
truehist(sample, main="Histogram of sample")
lines(x, y.w)
lines(x, y.g, col="red")
legend(30, 0.1, legend=c("Gamma distn", "Weibull distn"),
  lwd=1, col=c("red", "black"))
```



Some people prefer to look at cumulative density functions. For example:

```
# Draw cumulative density functions
plot(ecdf(sample), cex=0)
curve(pweibull(x, shape=fit.w$estimate[1],
  scale=fit.w$estimate[2]), add=T, col=2)
curve(pgamma(x, shape=fit.g$estimate[1],
  rate=fit.g$estimate[2]), add=T, col=3)
legend(30, 0.6, lwd=1, col=1:3,
  legend=c("Empirical CDF", "Weibull CDF", "Gamma CDF"))
```



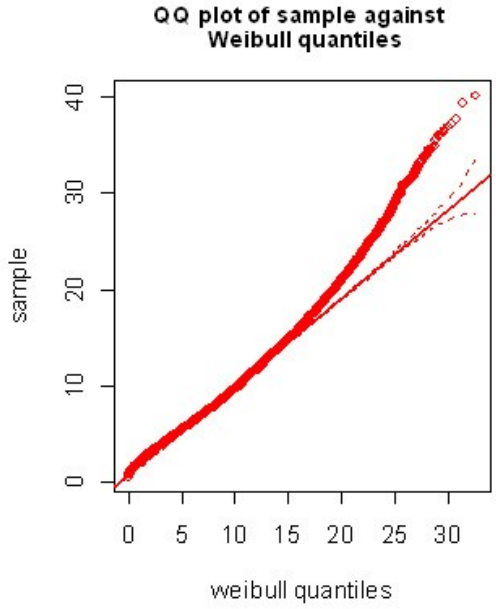
It is quite clear that the Gamma distribution is most appropriate for this data. However an alternative graphical measure is to consider a QQ-plot, which plots the empirical quantiles of the data to the theoretical quantiles of the fitted distribution. If the distribution is appropriate, we would expect the points to follow a straight line.

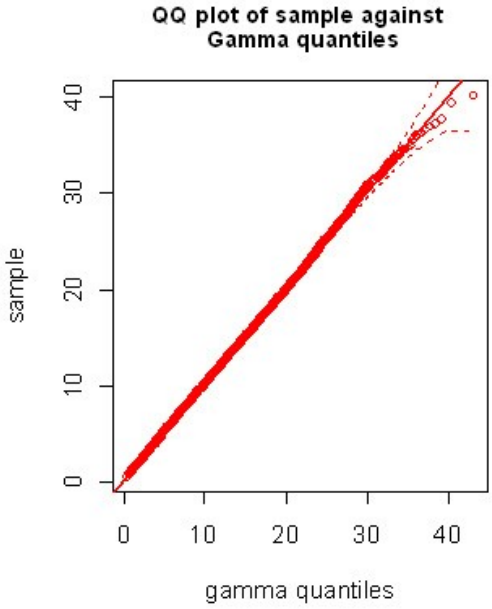
This is very simple to achieve manually, but the function `qq.plot` included in the `car` library is quite neat. For example:

```
library(car)
par(mfrow = c(1, 2))
qq.plot(sample, distribution="weibull",
  shape=fit.w$estimate[1], scale=fit.w$estimate[2])
title(main="QQ plot of sample against \n Weibull quantiles",
cex.main=0.9)
qq.plot(sample, distribution="gamma",
  shape=fit.g$estimate[1], rate=fit.g$estimate[2])
title(main="QQ plot of sample against \n Gamma quantiles",
cex.main=0.9)
```

The `par` command, sets the graphical parameter `mfrow`, so that we are able to draw two graphs side by side. See the help file for further details. As discussed in section 2.5, the "\n" character represents a new line.

In the resulting graphs, below, 95% confidence intervals are included around the expected straight line. As the points in the first graph clearly do not follow this line, the Weibull distribution is not appropriate. However, the second graph indicates that the Gamma distribution is indeed a good fit of the data.


QQ plot of sample against
Weibull quantiles
sample
0
10
20
30
40
0
5
10
15
20
25
30
weibull quantiles


QQ plot of sample against
Gamma quantiles
sample
0
10
20
30
40
0
10
20
30
40
gamma quantiles

3.4 Extreme Value Distributions

Extreme Value Theory (EVT), which as the name suggests is useful for modelling extreme events, has become more popular amongst actuaries in recent years. For example EVT might be useful when calculating an insurer's ICA, which by definition considers events at the 1 in 200 year level. It may also be helpful for pricing a high XL reinsurance layer. At the very least, EVT simply provides the actuary with an alternative set of distributions with which to model losses.

Two distributions for EVT are:

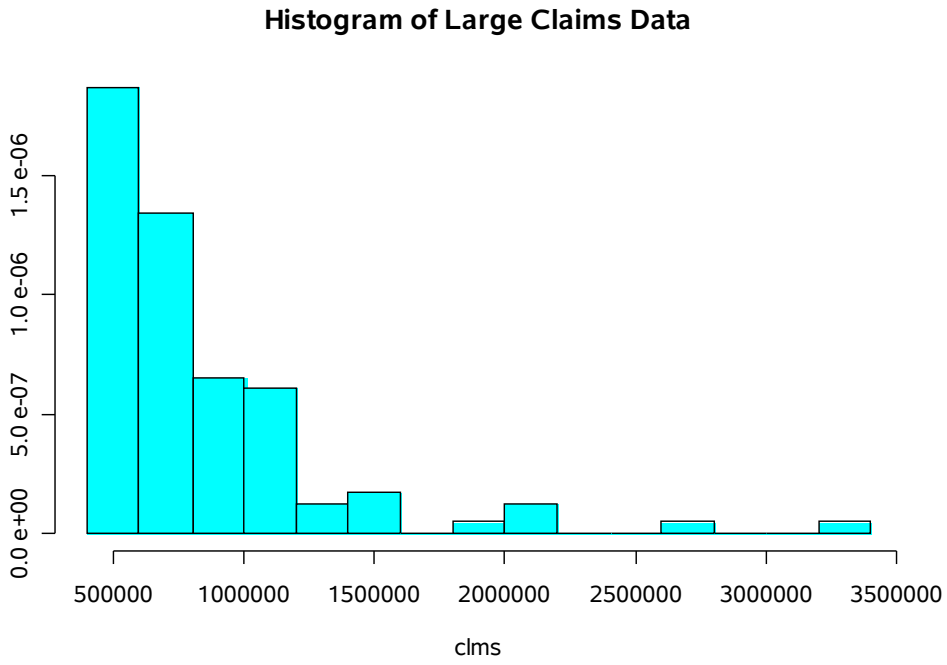
- Generalised Extreme Value (GEV) distribution. This family describes the distribution of the maxima of sets of observations, for example the largest annual claim on an insurance contract. It encompasses three classes of distribution: Gumbel (shape parameter, $\xi = 0$), Frechet ($\xi > 0$) and Weibull ($\xi < 0$).

- Generalised Pareto Distribution (GPD). This distribution describes exceedences over a threshold and is perhaps more convenient for modelling insurance claims.

Functions for fitting both distributions, including diagnostic functions are available in add on packages for R. We shall use functions from two such packages in this example. We shall consider a simple example, from "The Modelling of Extremal Events" by D Sanders (2005), which looks to fit a GPD.

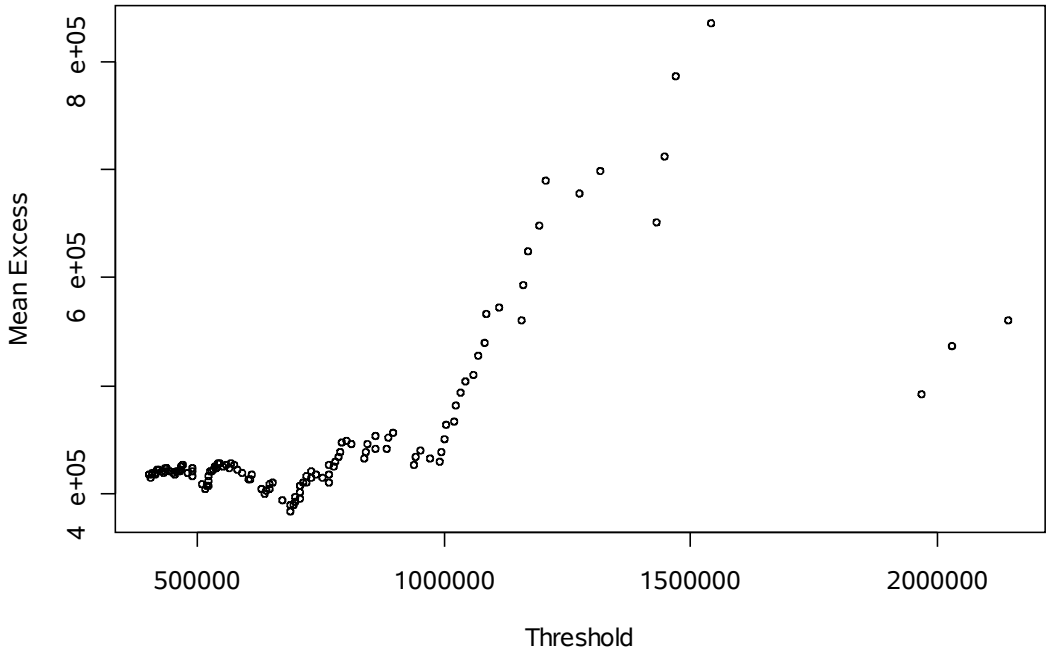
We start by loading up the data and viewing a histogram of the data

```
# Load up libraries to be used in analysis
library(evir)
library(evd)
library(MASS)

# Load up data
clms <- read.csv("C:\\R\\sanders.csv")
clms <- clms[,1]        # Want data in vector form

# Plot histogram
truehist(clms, nbins=20, main="Histogram of Large Claims Data")
```

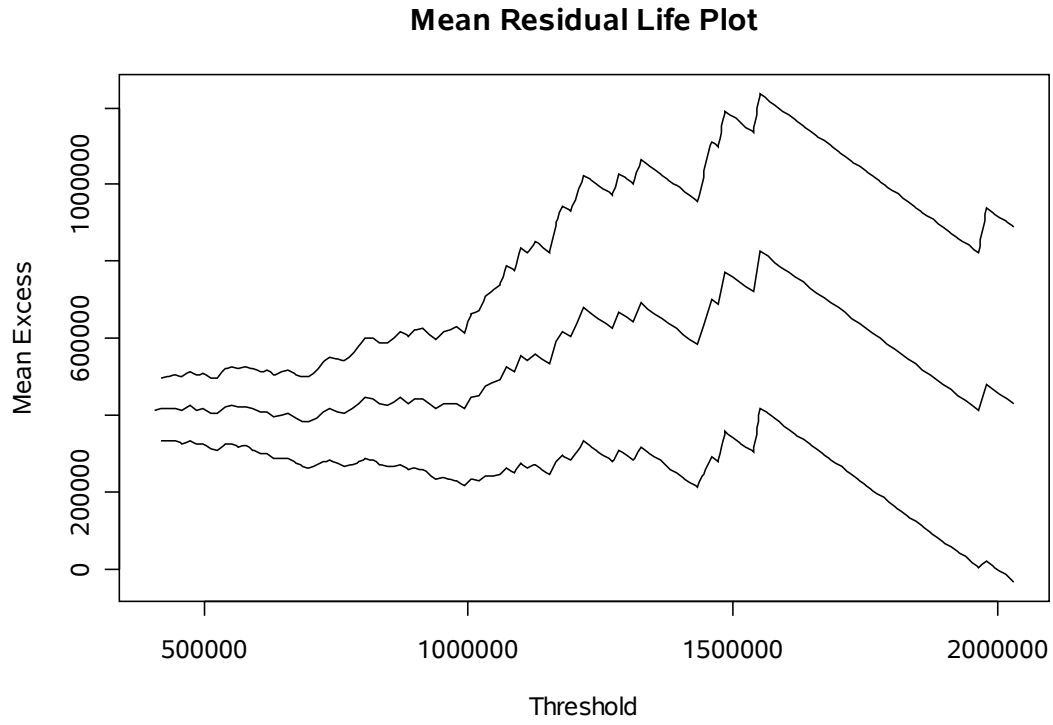


The GPD fits a distribution to exceedences over a threshold. We therefore need to pick an appropriate threshold. A useful plot to assist in this choice is the mean excess plot. The two libraries considered in this example both have their own functions to achieve this:

```
# Mean Excess Plot
meplot(clms)      # From evir package
```



```
mrlplot(clms)      # From evd package
```

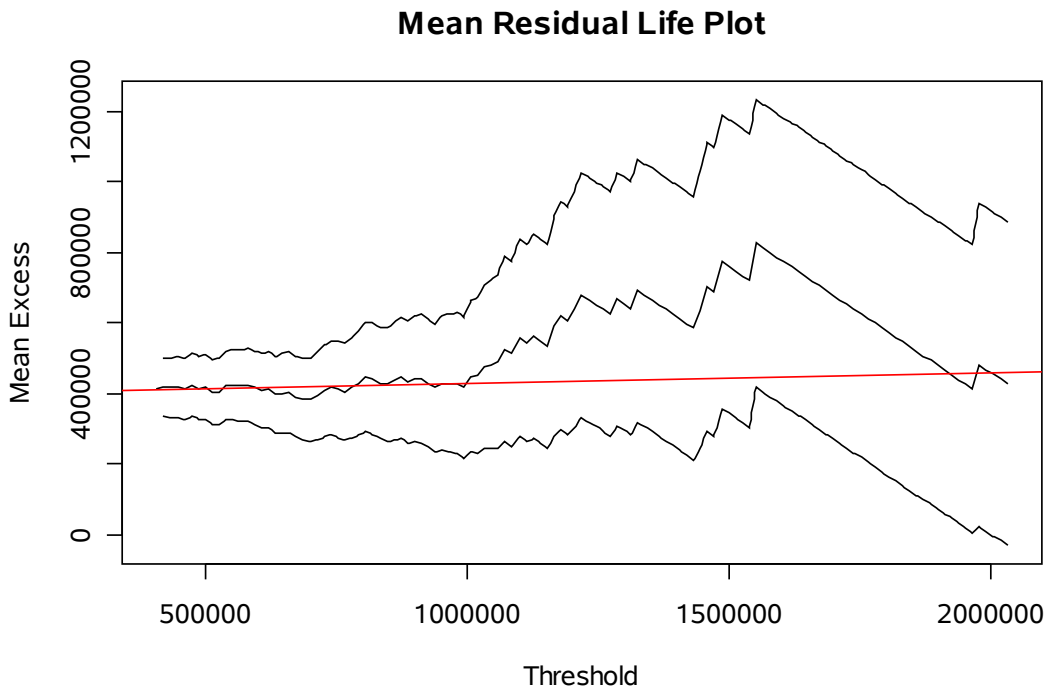


If the GPD is a good fit to the tail of the data, the plot should become approximately linear (before it becomes unstable due to the few very high data points). We want to pick the largest threshold beyond which a portion the graph starts to become linear. The higher the threshold, the better the GPD will approximate the tail. However, there will be fewer data points upon which to parameterize the distribution. Based upon the above graphs, one might choose a threshold of approximately 1,000,000 or perhaps a threshold of around 400,000. We examine the number of points that would be fitted in each case.

```
length(clms[clms>1000000])
[1] 28
length(clms[clms>400000])
[1] 123
```

The different slopes of the plot could indicate that different distributions need to be fitted. If we assume a threshold of 400,000, and were to fit a linear regression line to the linear portion following this point (from 400,000 to 1,000,000), we get the following

```
# Fit linear regression line (fitted to points < 1m)
# to mean excess plot
me <- u <- sort(clms)[-length(clms)]
for (i in 1:(length(clms)-1)) {
  data <- clms[clms > u[i]]
  me[i] <- mean(data - u[i])
}
fit <- lm(me ~ u, subset=u<1000000)
mrlplot(clms)
abline(fit, col=2)
```



As this falls within the 95% confidence interval of the above plot, we might conclude that the 400,000 threshold is reasonable. In fact, the original example used a threshold of 406,000, so we shall adopt the same. We fit the GPD distribution as follows, storing it in the variable `gpd.model`.

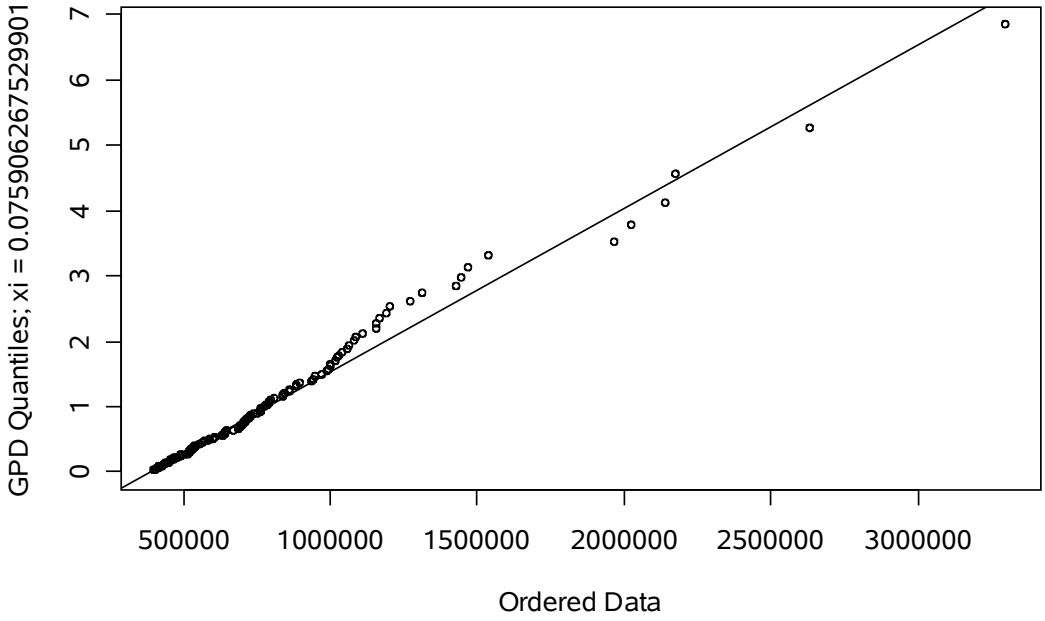
```
# Fit GPD
gpd.model <- gpd(clms, 406000)      # From evir package
```

We could view the output, by typing the name of the variable, but the output is too large to show here. Instead, we shall just examine the fitted parameters.

```
gpd.model$par.ests
          xi         beta
7.590627e-02 3.808000e+05
```

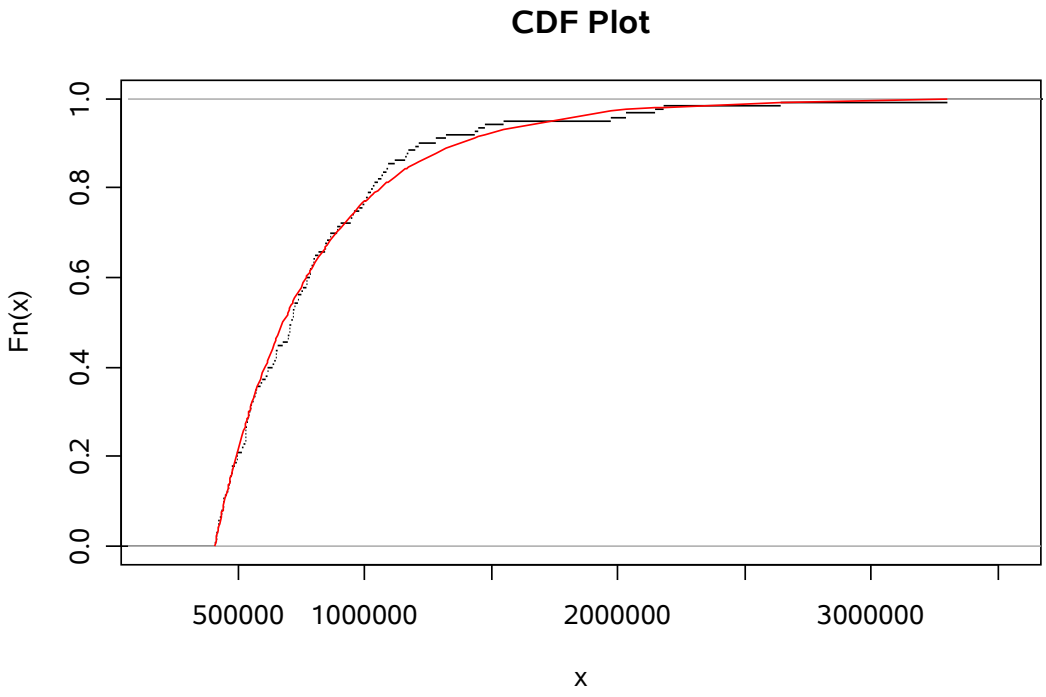
Useful diagnostics when fitting distributions are the QQ plot and the cumulative distribution function plots.

```
# QQ plot
qplot(clms, xi=gpd.model$par.ests[1])
```



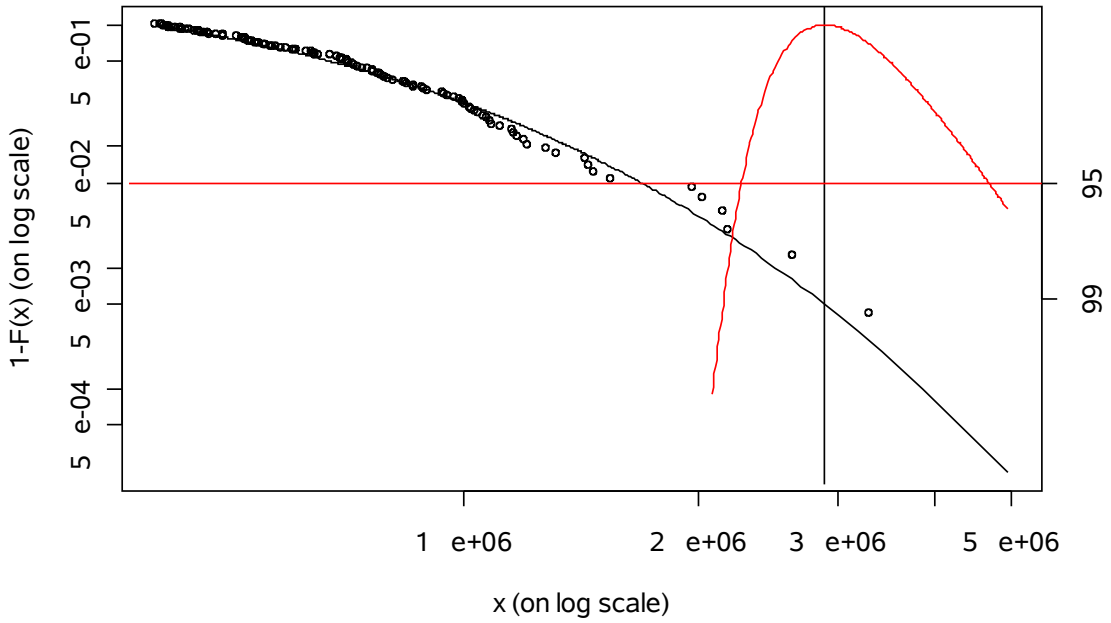
The straight line indicates that the resulting model is a good fit for the data.

```
# CDF Plot
plot(ecdf(clms), cex=0, main="CDF Plot")
lines(sort(clms), pgpd(sort(clms),
  xi=gpd.model$par.ests[1],
  beta=gpd.model$par.ests[2],
  mu=406000), col=2)
```



We might consider the worst loss on a 1 in 200 basis. We could do this simply, via the `qgpd` function. However, the standard function `gpd.q` provides a nice short alternative with a graph and confidence intervals:

```
gpd.q(tailplot(gpd.model), 0.995)
Lower CI Estimate Upper CI
 2266194  2889643  4689927
```



3.5 Generalized Linear Models – Stochastic Reserving

With the advent of Individual Capital Assessments (ICAs), Risk Based Capital, Solvency II, etc. much greater focus has been placed on understanding the uncertainty associated with claims reserves. The paper, Stochastic Claims Reserving in General Insurance by PD England and RJ Verrall, provides an excellent summary of many stochastic reserving methods. Some of these methods involve the use of Generalized Linear Models (or GLMs).

GLMs extend linear regression to accommodate both non-normal response distributions and transformations to linearity. An excellent introduction is provided in A Practitioner's Guide to Generalized Linear Models, by Duncan Anderson et al. (though the focus is on pricing).

This example, taken from England and Verrall's paper, considers the fitting of an over-dispersed Poisson distribution to the following claims data. It presumes some knowledge of stochastic reserving and GLMs, so please refer to the papers mentioned above.

Incremental Claims Data

	Development Period									
	1	2	3	4	5	6	7	8	9	10
Origin Period										
1	5012	3257	2638	898	1734	2642	1828	599	54	172
2	106	4179	1111	5270	3116	1817	-103	673	535	
3	3410	5582	4881	2268	2594	3479	649	603		
4	5655	5900	4211	5500	2159	2658	984			
5	1092	8473	6271	6333	3786	225				
6	1513	4932	5257	1233	2917					
7	557	3463	6926	1368						
8	1351	5596	6165							
9	3133	2262								
10	2063									

A complete discussion of this method and its formulation may be found in the relevant paper. It results in the following model:

$$E\left[C_{ij}\right]=m_{ij}=e^{\eta_{ij}}=e^{\alpha_i+\beta_j+c}$$

where

$$\mathrm{Var}\left[C_{ij}\right]=\varphi\, m_{ij}$$

C_{ij} = Incremental Claims Data
α_i = Origin Period i
β_i = Development Period j
c = Constant
φ = Dispersion Parameter
η_{ij} = Linear predictor

The over-dispersion manifests itself in the fact that the variance of the claims is *proportional* to the mean, rather than equal to it (as in the Poisson distribution). The log of the mean claim (in GLM-speak this means we are using a log link function) is

equal to a linear function of both the origin period and the development period. These are usually referred to as *factors*.
We start by loading up the data and

```
# Load up data
data <- read.csv("Mack.csv", header=F)
data <- as.matrix(data)
data
```

```
      V1   V2   V3   V4   V5   V6   V7  V8  V9 V10
1   5012 3257 2638  898 1734 2642 1828 599  54 172
2    106 4179 1111 5270 3116 1817 -103 673 535  NA
3   3410 5582 4881 2268 2594 3479  649 603  NA  NA
4   5655 5900 4211 5500 2159 2658  984  NA  NA  NA
5   1092 8473 6271 6333 3786  225   NA  NA  NA  NA
6   1513 4932 5257 1233 2917   NA   NA  NA  NA  NA
7    557 3463 6926 1368   NA   NA   NA  NA  NA  NA
8   1351 5596 6165   NA   NA   NA   NA  NA  NA  NA
9   3133 2262   NA   NA   NA   NA   NA  NA  NA  NA
10 2063   NA   NA   NA   NA   NA   NA  NA  NA  NA
```

We then need to prepare the data so that it is in a suitable form for fitting the GLM. We need the data to be in a vector format.

```
# Prepare data in correct form
claims <- as.vector(data)
```

Each data point is associated with a unique origin period and development period. We need to set up associated vectors for each factor, so that each element of the `claims` vector has a corresponding element in the `origin` and `dev` vectors:

```
n.origin <- nrow(data)
n.dev <- ncol(data)
origin <- factor(row <- rep(1:n.origin, n.dev))
dev <- factor(col <- rep(1:n.dev, each=n.origin))
```

We set up our origin and development period vectors using the `factor` function. When fitting the GLM, this will give each origin period and each development period its own parameter, as required for this model.

By putting these vectors together in a data frame, it is easier to see what is going on. It is not a necessary step for fitting the GLM.

```
# Put into a data frame (no need, but easier to visualise)
mack <- data.frame(claims=claims, origin=origin, dev=dev)
mack[1:5, ]       # Print first five rows
  claims origin dev
1   5012      1   1
2    106      2   1
3   3410      3   1
4   5655      4   1
5   1092      5   1
```

We see the claim value "5012" belongs to origin period 1 and development period 1. We are almost ready to fit our GLM, but first there is a technical issue to deal with. When fitting our GLM, we can implement the over-dispersed Poisson distribution using the `quasipoisson` family. In our example, the data contains a negative value. This is technically fine for a quasi-Poisson fit (provided the fitted mean is positive). Unfortunately, the standard `quasipoisson` function in R is overly restrictive. We need to adjust this function to suit our needs.

We can see the innards of any function by typing the name of the function with no subsequent brackets. For example:

```
quasipoisson
function (link = "log")
{
    linktemp <- substitute(link)
    if (!is.character(linktemp)) {
        linktemp <- deparse(linktemp)
        if (linktemp == "link")
            linktemp <- eval(link)
    }
    if (any(linktemp == c("log", "identity", "sqrt")))
        stats <- make.link(linktemp)
    else stop(gettextf("link \"%s\" not available for
quasipoisson family; available links are \"identity\", \"log\"
and \"sqrt\"",
        linktemp), domain = NA)
    variance <- function(mu) mu
    validmu <- function(mu) all(mu > 0)
    dev.resids <- function(y, mu, wt) 2 * wt * (y *
log(ifelse(y ==
        0, 1, y/mu)) - (y - mu))
    aic <- function(y, n, mu, wt, dev) NA
    initialize <- expression({
        if (any(y < 0)) stop("negative values not allowed for
the quasiPoisson family")
        n <- rep.int(1, nobs)
        mustart <- y + 0.1
    })
    structure(list(family = "quasipoisson", link = linktemp,
        linkfun = stats$linkfun, linkinv = stats$linkinv,
variance = variance,
        dev.resids = dev.resids, aic = aic, mu.eta =
stats$mu.eta,
        initialize = initialize, validmu = validmu, valideta =
stats$valideta),
        class = "family")
}
<environment: namespace:stats>
```

Don't get frightened by the result! In most situations there will not be any need to view the innards of standard functions. It is also possible that someone else has created a function suitable for your purposes. Therefore the internet is an invaluable resource.

In this case, the R-help archives provided the solution to just this problem, courtesy of David Firth. He provided the following code to "replace" the standard `quasiposson` function.

```
# New quasi-poisson family
quasipoisson <- function (link = "log")
## Amended by David Firth, 2003.01.16, at points labelled ###
## to cope with negative y values
##
## Computes Pearson X^2 rather than Poisson deviance
##
## Starting values are all equal to the global mean
{
     linktemp <- substitute(link)
     if (!is.character(linktemp)) {
         linktemp <- deparse(linktemp)
         if (linktemp == "link")
             linktemp <- eval(link)
     }
     if (any(linktemp == c("log", "identity", "sqrt")))
         stats <- make.link(linktemp)
     else stop(paste(linktemp, "link not available for
poisson",
         "family; available links are", "\"identity\", \"log\"
and \"sqrt\""))
     variance <- function(mu) mu
     validmu <- function(mu) all(mu > 0)
     dev.resids <- function(y, mu, wt) wt*(y-mu)^2/mu   ###
     aic <- function(y, n, mu, wt, dev) NA
     initialize <- expression({
         n <- rep(1, nobs)
         mustart <- rep(mean(y), length(y))             ###
     })
     structure(list(family = "quasipoisson", link = linktemp,
         linkfun = stats$linkfun, linkinv = stats$linkinv,
variance = variance,
         dev.resids = dev.resids, aic = aic, mu.eta =
stats$mu.eta,
         initialize = initialize, validmu = validmu, valideta =
stats$valideta),
         class = "family")
}
```

After running this piece of code, we are ready to fit our GLM.

```
# Fit model
model <- glm(claims ~ origin + dev, family = quasipoisson(),
  subset=!is.na(claims), data=mack)
```

We can examine a summary of the fitted model via:

```
summary(model)

Call:
glm(formula = claims ~ origin + dev, family = quasipoisson(),
    subset = !is.na(claims))

Deviance Residuals:
       Min          1Q      Median          3Q         Max
-4.728e+01  -2.120e+01   5.006e-14   1.566e+01   6.313e+01

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  7.65510    0.30921  24.757   <2e-16 ***
origin2     -0.11084    0.33406  -0.332   0.7420
origin3      0.24586    0.30838   0.797   0.4305
origin4      0.42134    0.30014   1.404   0.1689
origin5      0.42910    0.30300   1.416   0.1653
origin6      0.03481    0.34254   0.102   0.9196
origin7     -0.05932    0.36970  -0.160   0.8734
origin8      0.24319    0.36647   0.664   0.5112
origin9     -0.16027    0.49789  -0.322   0.7494
origin10    -0.02318    0.75658  -0.031   0.9757
dev2         0.69283    0.26000   2.665   0.0115 *
dev3         0.62603    0.26957   2.322   0.0260 *
dev4         0.27695    0.30164   0.918   0.3647
dev5         0.06056    0.33089   0.183   0.8558
dev6        -0.19582    0.37621  -0.521   0.6059
dev7        -1.08309    0.58871  -1.840   0.0741 .
dev8        -1.27366    0.76415  -1.667   0.1042
dev9        -1.91593    1.31937  -1.452   0.1551
dev10       -2.50760    2.41131  -1.040   0.3053
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for quasipoisson family taken to be
983.635)

    Null deviance: 84946  on 54  degrees of freedom
Residual deviance: 35411  on 36  degrees of freedom
AIC: NA

Number of Fisher Scoring iterations: 7
```

This output shows a summary of the model, with the fitted coefficients, their standard deviations, etc.

For stochastic reserving purposes, we have some more work to do. We start by extracting the relevant information from the model:

```
# Extract useful info from the model
coef <- model$coefficients          # Get coefficients
disp <- summary(model)$dispersion   # Get dispersion parameter
cov.param <- disp * summary(model)$cov.unscaled
# Get covariance matrix of parameters
```

Each element of the claims triangle is associated with a particular origin period and development period. Having used these to fit the model, we can then use the fitted coefficients to predict claims in future development periods. We need to make sure that we add the correct coefficients, for example to calculate the predicted claim cost for origin period 2, development period 10, we need to apply the following formula:

$E\left[C_{2,10}\right]=e^{\hat{\alpha}_2+\hat{\beta}_{10}+\hat{c}}$ where

¿ $e^{F\hat{\beta}}$

$\hat{\beta}$ = vector of fitted coefficients ($\hat{c}$, $\hat{\alpha}_i$ and $\{\hat{\beta}_j$)

¿ F = future design matrix ¿ ¿

The future design matrix is used to extract the relevant coefficients. In the above example, F would have a single row and nineteen columns (equal to the number of coefficients). All values would be zero except those corresponding to $\hat{c}$, $\hat{\alpha}_2$ and $\{\hat{\beta}_{10})$¿ .

In our example, we extend the above design matrix so that each row refers to a different future development period. The following code achieves this.

```
# To determine future uncertainty, need to create a
# design matrix for future payments. Build up in stages.
# Assume start from bottom left of future triangle.
n.fut.points <- length(claims[is.na(claims)])
fut.design <- matrix(0, nrow = n.fut.points, ncol=length(coef))

fut.points <- claims
fut.points[!is.na(claims)] <- 0
fut.points[is.na(claims)] <- 1:n.fut.points
for(p in 1:n.fut.points){
  # All points and a constant in the predictor
  fut.design[p, 1] <- 1
  # Row factor
  fut.design[p, 1 +
    as.numeric(origin[match(p, fut.points)]) - 1] <- 1
  # Col factor
  fut.design[p, 1 + (n.origin-1) +
    as.numeric(dev[match(p, fut.points)]) - 1] <- 1
}
```

We can then determine the expected claims in each future development period. Summing these gives us the expected total reserve.

```
# Determine fitted future values (as a diagonal matrix)
fitted.values <- diag(as.vector(exp(fut.design %*% coef)))
total.reserve <- sum(fitted.values)
total.reserve
[1] 52135.23
```

Note that we have expressed the fitted values as a diagonal matrix. This will assist with the variance calculations that come next. The result above actually corresponds to the result following a standard chain ladder reserving method.

The England and Verrall paper, provides the following approximation for the mean squared error of prediction for the entire reserve.

$$MSEP\left[\hat{C}_{++}\right] \approx \sum_{i,j \in \Delta} \varphi m_{ij} + \sum_{i,j \in \Delta} \hat{m}_{ij}^2 \operatorname{Var}\left[\hat{\eta}_{ij}\right] + 2 \sum_{\substack{i_1,j_1 \in \Delta \\ i_2,j_2 \in \Delta \\ i_1,j_1 \neq i_2,j_2}} \hat{m}_{i_1,j_1} \hat{m}_{i_2,j_2} \operatorname{Cov}\left[\hat{\eta}_{i_1,j_1} \hat{\eta}_{i_2,j_2}\right]$$

We take the square root of this using the following code to perform the relevant matrix operations.

```
# Determine covariance matrix of linear predictors
cov.pred <- fut.design %*% cov.param %*% t(fut.design)

# Determine covariance matrix of fitted values
cov.fitted <- fitted.values %*% cov.pred %*% fitted.values

# Determine uncertainty statistics
total.rmse <- sqrt(disp*total.reserve+sum(cov.fitted))
total.predictionerror <- round(100*total.rmse/total.reserve)

total.rmse
[1] 17612.73

total.predictionerror
[1] 34
```

Therefore, our model suggests that the expected reserve is 52,135, with a standard error of prediction of 17,613 (or 34%).