Working with R

An introduction to R and RStudio

Miles Benton

Last Updated 1st October 2012

Contact: miles.benton84@gmail.com

(page left intentionally blank)

Contents

Preface

This document is a collection of notes and useful titbits that I've hoarded over the last year and a bit since I've been learning R. I just want to reiterate that I am by no means an expert, and am constantly picking up new methods and techniques throughout my Phd. The great thing about R (and, as I'm quickly learning, most programming languages) is there are many ways to achieve the same end point. More often than not when you're starting out it's about what's easy to grasp and what actually works, as opposed to beautiful code and elegant design. Yes, sure there may well be a 'better' way of doing things, but if you've managed to stumble your way through a project and have got an end result then congratulations; give yourself a pat on the back! ☺

Enough about me, *'what's the deal with this workshop?'* you're asking. Well since bioinformatics and high-throughput genomics applications are now quickly becoming ingrained in many labs, and data sets are expanding in size at a phenomenal rate – it is not uncommon to see some next gen sequencing data in the terabytes – there needs to be efficient and powerful ways to mine and analyse this data. R just happens to lend itself to this purpose, and is quickly becoming one of the major go-to tools for bioinformatics analysis. This in mind, R is still a programming language and generally these aren't the easiest things to just pick up and learn.

So that's where this workshop comes in; this is the first GRC/Griffith University workshop for those interested in applications of R In medical research, bioinformatics and high-throughput genomics. Somewhat of a pilot, the aim is to offer an initial grounding in R and set the platform for possible future workshops that will deal with handling and analysis of genomic data, i.e. microarrays. I have attempted to make this document as easy to follow as possible, and hope that those attending (and others who were unable to make it) will be able to take it away and use it as a reference and tutorial they can work through at their own pace.

This document is broken into numerous sections[1]. The first couple offer information on where to get R/RStudio, how to install, and getting started. I've also included information about resources I've found invaluable whilst learning (in most cases these will have an html link). The actual workshop will be based on a core of four sections:

1. A Brief Introduction to R Language and Programming
2. Data Management in R
3. Basic Statistical methods
4. Basic Plotting/Graphing in R

You might notice that these four sections are generally lighter on text – and contain much more code – than the initial 'introduction' of this document. This is because I will be talking around the code and examples as we go. I have included some barebones comments and explanations that you can hopefully refer back to at a later date if needed. Also after each section there is an exercise that aims to reinforce the ideas covered previously. At the very end of the document you'll find a 'cheat sheet' that details some of the more common and useful functions/parameters utilised in R.

Well that's a brief synopsis of how things are planned to proceed, so good luck and I hope you enjoy.

Miles

[1] I thought about using chapters, but it just made me realise how large this 'manual' was becoming! This is as good a time as any to point out that there will be several footnotes – with some containing important information (as well as being a tribute to the great Terry Pratchett – a staunch proponent of the footnote).

What is R?

R is a programming language and software environment for statistical computing and graphing. It is an implementation of the S language, was created at the University of Auckland in New Zealand, and is now developed by the *R Development Core Team*. R is part of the GNU project with its source code freely available under the GNU General Public Licence, and precompiled binary versions[2] available for various operating systems. R uses a command line interface, yet there are many groups developing and maintaining graphical user interfaces (Gui's) for use with R.

R provides a wide variety of statistical techniques; linear and non-linear modelling, classical statistical tests, time-series analysis, classification, clustering, and more. It also features powerful graphing techniques and is high extensible through installable packages. R provides a powerful means for research in statistical methodology and bioinformatics.

The R environment

R is an integrated suite of software facilities for data manipulation, calculation and graphical display. It includes:

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

Why use R?

1. **It's free.**
   R is open source software, and as such is free to everyone. There is a very active community of developers that constantly work on add functionality and advancing R.

2. **It runs on a variety of platforms, including Windows, Linux and Mac OS X.**
   So whatever your flavour you'll be able to run it.

3. **It provides an unparalleled platform for programming new statistical methods in a fairly easy and straight forward manner.**
   Due to the package nature of R it's fairly easy to develop new tools quickly.

4. **It contains advanced statistical routines not yet available in other software.**

5. **It has state of the art graphics capabilities.**
   With R you have complete control over plotting/graphing options and are able to output publication quality figures with ease.

6. **It's fun![3]**

Where can I get R?

R can be obtained from the r-project website (http://cran.r-project.org). For a much more detailed explanation of how to get R and install it please read on.

---

[2] Don't worry if this makes no sense, it's just fancy language for 'click to install'.

[3] Your mileage may vary, but hopefully it will become less of a chore and more enjoyable as you learn.

A Collection of Useful Resources

There is a multitude of R resources available online for both learning and finding answers to questions. Here I have compiled a brief list to get you started, if you find other resources that are helpful make sure to record them and share with the group.

Required software

Below are links to the R project and RStudio websites.

- R: www.r-project.org/
  - » Windows: http://cran.r-project.org/bin/windows
  - » Linux: http://cran.r-project.org/bin/linux
  - » Mac OS X: http://cran.r-project.org/bin/macosx

- RStudio: http://rstudio.org/

Package Repositories

There are well over 3000 packages available from CRAN but navigating them all can be a challenge. Below I have listed the main package sites, but have also included several resources that help you to locate the most appropriate package for the task at hand.

- CRAN: http://cran.r-project.org/
  - » CRAN Packages: http://cran.r-project.org/web/packages
  - » CRAN Task Views: http://cran.r-project.org/web/views

- Crantastic: http://crantastic.org

A community driven site for R packages where you can search for, review and tag CRAN packages.

- Bioconductor: http://www.bioconductor.org/

Open source and development tools for bioinformatics and high-throughput genomic analysis.

Learning R

If you are learning R there are many good places to start. The following are just a few examples of nice resources, introductions and tutorials for beginners (**highlighted** are my favourites).

- Basic R tutorial: http://www.cyclismo.org/tutorial/R
- **CRAN introduction to R:** http://cran.r-project.org/doc/manuals/R-intro.pdf
- **The R reference card:** http://cran.r-project.org/doc/contrib/Short-refcard.pdf
- **Quick-R:** http://www.statmethods.net/index.html
- CRAN Contributed Documentation: http://cran.r-project.org/other-docs.html
- Free Course: http://oli.web.cmu.edu/openlearning/forstudents/freecourses/statistics

Asking Questions

A great place to start your search for answers is RSeek (http://www.rseek.org), a search engine which provides a unified interface for searching the various sources of online R information. If there is an existing answer to your question it is highly likely that you'll find it through RSeek.

If you can't find answers with RSeek you could try:

- Stack Overflow: http://stackoverflow.com/questions/tagged/r
- The R-help mailing list: https://stat.ethz.ch/mailman/listinfo/r-help

Installation

Installation of R is fairly straight forward, although it differs slightly between platforms. All platforms have binaries (executable installers) available, and these are recommended for most users. If you want to build R from source you'll need the suitable compilers depending on your platform.[4]

Windows

The `bin/windows` directory of a CRAN site contains binaries for a base distribution and a large number of add-on packages from CRAN to run on Windows XP or later on ix86 CPUs (including AMD64/Intel644 chips and Windows x64). You can locate the above directory via this link: http://cran.r-project.org/bin/windows/base/

Installation is via the installer – "**R-2.15.1-win.exe**" is current stable release. Once downloaded just double-click on the icon and follow the instructions. When installing on a 64-bit version of Windows the options will include 32- or 64-bit versions of R (and the default is to install both). You can uninstall R from the Control Panel or from the (optional) R program group on the Start Menu.

Linux

At the time of writing there are four Linux distributions that have R binaries available; `debian`, `redhat`, `suse` and `Ubuntu`. They are located here: http://cran.r-project.org/bin/linux
I'm not sure if these binaries would work under alternate distributions (likely in most cases, but your mileage may vary).

The R source code can be downloaded and compiled with little hassle on most Linux distros. A guide for compiling R can be found here: http://cran.r-project.org/doc/manuals/R-admin.pdf

Example installation under Debian/Ubuntu

To install the complete R system, use:

```
sudo apt-get update
sudo apt-get install r-base
```

Users who need it can also install the r-base-dev package (development release):

```
sudo apt-get install r-base-dev
```

Mac

There is a nice FAQ on R for Mac OS X users which is frequently maintained and updated regularly. It can be found here: http://cran.r-project.org/bin/macosx/RMacOSX-FAQ.html

The `'bin/macosx'` directory of a CRAN site contains binaries for OS X for a base distribution and a large number of add-on packages from CRAN to run on OS X 10.5/6/7.

The simplest way to install R on Mac is to use `'R-2.15.1.pkg'`: just double click on the icon.
Note: `'R-2.15.1.pkg'` is the latest stable R release at time of writing.

[4] If you are using Linux you'll be good to go, for Windows you'll need to obtain a program to compile source code, MinGW (http://www.mingw.org) and Cygwin (http://www.cygwin.com) are both good for this.

Installing Bioconductor

"Bioconductor provides tools for the analysis and comprehension of high-throughput genomic data. Bioconductor uses the R statistical programming language, and is open source and open development. It has two releases each year, more than 460 packages, and an active user community."

We won't cover the use of Bioconductor in this workshop, but I've included some information to get you started if you are interested in exploring Bioconductor and its packages.

Use the `biocLite.R` script to install Bioconductor packages. To install a particular package, e.g., `limma`, type the following in an R command window:

```
source("http://bioconductor.org/biocLite.R") # downloads the latest bioconductor script/repos
biocLite("limma")
```

To install a selection of core Bioconductor packages, use:

```
biocLite()
```

Packages and their dependencies installed by this usage are: `affy, affydata, affyPLM, affyQCReport, annaffy, annotate, Biobase, biomaRt, Biostrings, DynDoc, gcrma, genefilter, geneplotter, GenomicRanges, hgu95av2.db, limma, marray, multtest, vsn,` and `xtable`. After downloading and installing these packages, the script prints "Installation complete" and "TRUE". **Warning: there are some large packages, the whole set might take a while to download.**

Checking and updating "old" packages

There are several ways that this may be achieved:

1. Auto-update in RStudio

This is the easiest way to update and maintain your packages. A package update option has been included since version 94.XX, just go to the packages tab and select search for updates. This will open a window and display which (if any) packages have newer versions available. This works for both CRAN and Bioconductor packages.[5]

2. Via console

Bioconductor packages, especially those in the development branch, are updated fairly regularly. To identify packages requiring update, start a new session of R and enter:

```
# bioconductor
source("http://bioconductor.org/biocLite.R")
old.packages(repos = biocinstallRepos())
# CRAN packages
old.packages()
```

To update all installed packages that are out of date enter:

```
# bioconductor
update.packages(repos = biocinstallRepos(), ask = FALSE, checkBuilt = TRUE)
# CRAN packages
update.packages()
```

[5] **Note:** This ability appears to change between RStudio releases. As of the current release (0.96.331) it is working.

R Development version

A note on using development versions:

There are development versions of R and Bioconductor available for public use. These are nightly builds and as such are not fully stable (**read: likely to contain bugs!**). However they contain the most up-to-date version of the software and may have specific functions and features that aren't present in the current stable release.[6]

For example the most recent stable R build is 2.15.1, while the development version is currently at 2.16.0 (the Developer's aim to release an updated stable version of R yearly). Bioconductor releases follow a 6 monthly release cycle; at the moment Bioconductor stable is 2.10, development is at 2.11 and is expected for stable release in October 2012. If you are interested in using R and Bioconductor to analyse microarray data that was generated on newer technology, you'll more than likely require development versions of the software to access all the analytical functionality as it is updated frequently by package developers.

To install the developer's version of R visit the development section of the official site and follow the links to the required build (i.e. Windows, Linux, Mac). Follow the instructions detailed above to install.

Once you have the development version of R installed you'll be able to install the latest Bioconductor development release using the same code as detailed above (see section "Installing Bioconductor").

Current development links

Windows 32/64bit:
http://cran.r-project.org/bin/windows/base/rdevel.html
Note: Windows versions can be compiled via source, though you'll need a gcc compatible compiler – if this is all jargon to you just stick with the binary installers. ☺

Linux 32/64bit:
http://cran.r-project.org/bin/linux/ (you'll have to choose flavour, download via `sudo`, or compile from source)

Mac 32/64bit:
http://r.research.att.com/

Troubleshooting

For some issues that arise you need to ask the mailing list for help. Remember to include a detailed description of your problem, along with the output of these two functions:

`traceback()` prints the call stack of the last uncaught error, i.e., the sequence of calls that lead to the error. This is useful when an error occurs with an unidentifiable error message.

`sessionInfo()` collects information about the current session and prints version information about R and attached or loaded packages.

[6] It's highly likely that you'll have to download 'nightly' builds of R quite regularly as many Bioconductor developers use these when writing their packages. Fear not, this isn't as gruelling as it may sound.

An introduction to RStudio

"RStudio is a new integrated development environment (IDE) for R. RStudio combines an intuitive user interface with powerful coding tools to help you get the most out of R."

RStudio is an IDE that works with the standard version of R that is freely available from CRAN. Like R, RStudio is available under a free software license. The goal of RStudio is to provide a powerful tool that is easy for beginners to grasp and at the same time offers increased productivity for advanced users.

I chose RStudio for this workshop as I believe it makes R easier to understand and operate, and allows an easier transition from the typical gui environment that most users are used to.[7] It has features such as syntax highlighting and code completion that aid in the learning of the R language, and the object and history browser allow the user to see and understand how R is working behind the scenes – you can even convince yourself that your data is loaded by clicking on it and seeing a nice spread sheet like view!

There are of course many alternatives if you decide you don't like RStudio, or just feel like trying something different. A list of gui projects is available here: http://sciviews.org/_rgui. R Commander is apparently very good and will be quite familiar for people who have used SPSS, link here: http://socerv.mcmaster.ca/jfox/Misc/Rcmdr

This following section of the workshop will guide you through installation and the basics of setting up of, and working within, an R/RStudio environment. The sections after this will deal mainly with specific R coding and functions, and as such if anything RStudio specific arises I will cover it at the time. I'm hoping that RStudio should come to feel like a natural extension of R and not as two separate programs.

Download and Installation

RStudio is available for Windows, Linux and Mac OS X (10.5+). The software can be downloaded from http://rstudio.org/download/desktop. This page will attempt to determine the best download for your system, but you may also choose from the list below.

Like R, there are precompiled binary install files available for most popular OS. If yours is not listed, and none of the binaries work, then you should download the source code and compile RStudio on your system.

There is also a nice option to download a zip/tarball version of RStudio. This version doesn't require installation, so if you plan to use it on a system where you don't have admin/root privileges this is the one you want to download. This version is also ideal to install on USB data sticks – giving you a portable version that you can travel with.[8]

Note: RStudio is still in beta, thus you may come across an error or two. However updates are released frequently and the community are only too happy to help out. I have been running RStudio on Windows XP/7/Server 2008, Ubuntu and Debian with zero issues as yet. Current stable version is v0.96.331 – as of the 1st Oct 2012.

[7] RStudio however is not a front end gui for R.
[8] R can also be installed to a USB drive, so you can have a completely portable analysis system at all times.

Using different versions of R within RStudio

If you recall from the previous section I mentioned development versions of R. By nature these releases aren't fully stable and thus the majority of users will stay away from them. However sometimes a development build is required by certain packages you will use. But what if you don't want to uninstall your current stable version of R? Well you don't have to. You can have multiple versions of R installed on the same system[9], and just use the version you require each session.

RStudio allows you to work with different versions of R as well. To start RStudio with a specific version all you need to do is hold down `[Ctrl]` while starting RStudio (double clicking the executable – again I've tested this in Windows and Linux, can't speak for Mac). This will open a small window where you can choose which installed R version you'd like to run inside RStudio.

Customisation Options

You can access RStudio options from the Tools | Options menu which includes the following sections:

- General R Options – default CRAN mirror, initial working directory, workspace and history behaviour.
- Source Code Editing – enable/disable line numbers, selected word and line highlighting, soft-wrapping for R files, parenthesis matching, right margin display, and console syntax highlighting; configure tab spacing, set default text encoding.
- Appearance and Themes – specify font size and visual theme for the console and source editor.
- Pane Layout – locations of console, source editor, and tab panes; set which tabs are included in each pane.

Navigating RStudio

There are four working panes in RStudio:

- **Source code pane**
  This pane is where scripts are written/loaded and displayed. It has syntax highlighting and auto completion, and allows you to pass code line by line, as selections, or as a whole.

- **R console pane**
  This is where commands are executed and is essentially what the base R console looks like, except better! The console has syntax highlighting, code completion and interfaces with the other RStudio panes.

- **Workspace/History pane**
  The workspace tab displays information that is usually hidden in R, such as loaded data, functions and other variables. The history tab stores all the commands (lines of code) which have been parsed through R.

- **Everything else pane**[10]
  This pane includes the files tab (lists all files in current working directory), the plots tab (any plots/graphs), packages tab (installed packages, and the help tab (inbuilt html help system).

[9] Install each version as you would normally (see instructions in the previous section).
[10] For lack of a better name…

Working Directories, .RData and .Rhistory

The working directory is the location that will contain the files you want to analyse, any scripts specific to the analysis, and will usually be the place where your output will be saved.[11] R sessions may be saved and loaded for future use using the .RData file structure. The default behaviour of R for the handling of .RData files and workspaces encourages and facilitates a model of breaking work contexts into distinct working directories. This section describes the various features of RStudio which support this workflow.

Default Working Directory

As with the standard R GUI, RStudio employs the notion of a global default working directory. Normally this is the user home directory (typically referenced using ~ in R). When RStudio starts up it does the following:

- Executes the .Rprofile (if any) from the default working directory.
- Loads the .RData file (if any) from the default working directory into the workspace.
- Performs the other actions described in R Startup.

When RStudio exits and there are changes to the workspace, a prompt asks whether these changes should be saved to the .RData file in the current working directory.

This default behaviour can be customized in the following ways using the RStudio Options dialog:

- Change the default working directory
- Enable/disable the loading of .RData from the default working directory at start-up
- Specify whether .RData is always saved, never saved, or prompted for save at exit.

Changing the Working Directory

The current working directory is displayed by RStudio within the title region of the Console. There are a number of ways to change the current working directory:

- Use the setwd R function (covered in the next section)
- Use the 'Session | Set Working Dir' menu. This will also change directory location of the Files pane.
- From within the Files pane, use the 'More | Set As' Working Directory menu. (Navigation within the Files pane alone will not change the working directory.)

Be careful to consider the side effects of changing your working directory:

- Relative file references in your code (for datasets, source files, etc) will become invalid when you change working directories.
- The location where .RData is saved at exit will be changed to the new directory.

Because these side effects can cause confusion and errors, it's usually best to start within the working directory associated with your project and remain there for the duration of your session.

Note: *We will return to the concept of working directories in latter sections when we start working with R, but just keep in mind that in R there are always numerous ways to achieve the final goal.*

[11] All these things can be sourced from other directories; however it is good practice to have them in the working directory as this makes your analysis completely reproducible.

Working with .RData files

As mentioned above, R sessions may be saved and loaded using .RData files.[12] It is good practice to keep these files in their respective working directories for ease of use. There are several ways to save and load sessions within R/RStudio.

How can I save my work?

You can save all the objects and functions that you have created in an .RData file, by using the R console and the `save()` or the `save.image()` functions. It is very important that you remember to include the .RData extension when indicating the file path because R will not supply it for you!

```
save(file = "d:/file_name.RData")     # saves R session to file_name.RData
save.image("d:/file_name.RData")      # saves R session to file_name.RData
```

RStudio allows you to do this through the file menu:
Workspace | Save Workspace As... - browse to the folder where you want to save the file and enter the file name of your choice

How can I retrieve the work that I have saved using a save.image function?

The `load()` function will load an .RData file.

```
load("d:/file_name.RData")     # loads R session from file_name.RData
```

RStudio allows you to do this through the file menu:
Workspace | Load Workspace... - browse to the folder where you saved the .RData file and click open

Handling of .Rhistory

The .Rhistory file determines which commands are available by pressing the up arrow key within the console. By default, RStudio handles the .Rhistory file differently than the standard R console or GUI, however can be configured to work the same was as those environments if you wish.

The conventional handling of .Rhistory files is as follows:

- Load and save .Rhistory within the current working directory
- Only save the .Rhistory file when the user chooses to save the .RData file

Whereas the default RStudio handling of .Rhistory files is:

- Load and save a single global .Rhistory file (located in the default working directory)
- Always save the .Rhistory file (even if the .RData file is not saved)

The RStudio defaults are intended to make sure that all commands entered in previous sessions are available when you start a new RStudio session. If you prefer the conventional R treatment of .Rhistory files you can customize this behaviour using the General panel of the Options dialog.

Note: You can also copy code from the history pane across into either the source pane or the console pane using the To Console or To Source buttons. This can be very useful, for example, if you are creating a script to run batch analysis on further data, or you want to save a portion of analysis for later use.

[12] .RData files store all objects that are loaded in the workspace at time of saving.

How do I save all the commands that I have used in an R session?
You can save a history of your R session in a .Rhistory file by using the `history()` function. It is very important that you remember to include the .Rhistory extension when indicating the file path because R will not supply it for you!

```
history("d:/file_name.Rhistory")      # saves .Rhistory file
```

In RStudio this can be achieved via the 'save history' button in the history tab.

How can I retrieve the work that I have saved using a history function?
The `loadhistory` function will load a .Rhistory file.

```
loadhistory("d:/file_name.Rhistory")          # loads .Rhistory file
```

In RStudio this can be achieved via the 'load history' button in the history tab.

Code Completion: Tab becomes your best friend

Using the `[Tab]` key allows you to auto complete or search for files/functions/data and much more. If you have used a Unix/Unix-like environment you will probably be very familiar with this functionality.

Example: type `lib` into the R console pane and then press `[Tab]`. What are you presented with?

This also works for files that are located in your home directory. Type `""` (quote marks) into the console and press `[Tab]`, you can start to see how useful this function is. Many programmers will probably call me lazy, but I find using `[Tab]` invaluable – it saves a heck of a lot of typing!

Retrieving Previous Commands

As you work with R you'll often want to re-execute a command which you previously entered. As with the standard R console, the RStudio console supports the ability to recall previous commands using the arrow keys:

- `[Up]` - Recall previous command(s)
- `[Down]` - Reverse of Up

If you wish to review a list of your recent commands and then select a command from this list you can use `Ctrl+Up` to review the list (note that on the Mac you can also use `Command-Up`).

RStudio Library and Package Management

RStudio has a tab dedicated to R packages and their management. Here you can see what packages are installed in your library(s) and which ones are in use. Packages can be 'loaded' and 'unloaded' using the check box on the left hand side. You can also 'click' on the package name to load a help section dedicated to that package (includes pdf manual, vignettes, and lists of functions/examples). Packages can also be updated from this tab by selecting the `"Check for Updates"` button.

In Windows, RStudio also gives you the option of creating a separate library in your user account directory (usually something like: `"C:\Users\username\Documents\R\win-library\2.15"`). This will become the default location for packages to be installed and mitigates many issues that arise when using R on a system that you haven't got admin/root privileges on.

We will explore using the library and packages in more detail in the next section.

Getting Help

If you require help or have any questions about RStudio there are several places you can head for information:

- RStudio Documentation: http://rstudio.org/docs
  There is a listing of many different documents available to browse, including sections on; using RStudio, RStudio Server and Advanced Topics.

- RStudio Support and Knowledge Base: http://support.rstudio.org/
  If you are having more of a technical problem this is the place to search for the answers, or if you can't find them, post your issue. The community is very active and posts are read and answered incredibly quickly.

If you require help with R related issues see the inbuilt html R Help documents[13], or the previous section of this document.

[13] To open R help use either `?fun` or `help(fun)`; where `fun` is the function/command/parameter you are looking for help on.

RStudio Keyboard Shortcuts

Below I have included a list of useful RStudio keyboard shortcuts for Windows/Linux (for a detailed list of keyboard shortcuts and Mac keys visit http://rstudio.org/docs/using/keyboard_shortcuts).

Console	
Clear console	Ctrl+L
Popup command history	Ctrl+Up
Yank line up to cursor	Ctrl+U
Yank line after cursor	Ctrl+K
Insert currently yanked text	Ctrl+Y
Source	
New Document	Ctrl+Shift+N
Close active document	Ctrl+Shift+L
Run current line/selection	Ctrl+Enter
Run current document	Ctrl+Shift+Enter
Switch to tab	Ctrl+Alt+Down
Previous tab	Ctrl+Alt+Left
Next tab	Ctrl+Alt+Right
First tab	Ctrl+Shift+Alt+Left
Last tab	Ctrl+Shift+Alt+Right
Extract function from selection	Ctrl+Alt+F
Comment/uncomment	Ctrl+Shift+C
Delete Line	Ctrl+D
Move Lines Up/Down	Alt+Up/Down
Copy Lines Up/Down	Ctrl+Alt+Up/Down
Select Page Up/Down	Shift+PageUp/PageDown
Select to Start/End	Shift+Alt+Up/Down
Completions (Console and Source)	
Attempt completion	Tab or Ctrl+Space
Navigate candidates	Up/Down
Accept selected candidate	Enter, Tab, or Right
Show help for selected candidate	F1
Dismiss completion popup	Esc
Views	
Move cursor to source	Ctrl+1
Move cursor to console	Ctrl+2
Show workspace	Ctrl+3
Show data	Ctrl+4
Show history	Ctrl+5
Show files	Ctrl+6
Show plots	Ctrl+7
Show packages	Ctrl+8
Show help	Ctrl+9
Plots	
Previous plot	Ctrl+PageUp
Next plot	Ctrl+PageDown

***Note:** newer releases of RStudio have resulted in a few of these shortcuts being depreciated.

(page left intentionally blank)

Section1: A brief introduction to R language and programming

This is the first of the workshop sections. Here we begin to explore the console environment of R and start touching on some of its capabilities. I will also spend a bit of time explaining the library and package system that is used by R.

In contrast to many other statistical analysis packages, analysis in R is not based on graphic user interface, but is command line-based. When you first start R, a command prompt appears. To get help and overview of R, type `help.start()` on the command line and press enter. This will start your internet browser (or if working in RStudio, the help tab) and open the main page of the R documentation. If you ever need further help in R you can type either `help(fun)` or `?fun`, where `fun` is a function or term you want to search for help on.

Let us first start off by using R as a calculator.

R as a calculator[14]

You can directly operate with numbers in R.

Summation

```
>2+4
[1] 6
```

Subtraction

```
>2-4
[1] -2
```

Multiplication

```
>2*4
[1] 8
```

Division

```
>2/4
[1] 0.5
```

Power

```
>2^4
[1] 16
```

Mathematical functions, such as square roots, base-10 logarithm, and exponentiation, are available in R as well:

```
>sqrt(16)
[1] 4
>log10(4)
[1] 0.60206
>exp(0.60206)
[1] 1.825876
```

The above operations and functions may be nested to achieve the same result in one line of code.

```
>exp(log10(sqrt(2^4)))
[1] 1.825876
```

[14] **A quick note about example code in this document.** As you can see, I've formatted the R code in an attempt to highlight some of the syntax, and portray how it will look in your R console. In the above I've used `>` (this is the R prompt) and tried to replicate what you'll see when working – the rest of the document doesn't use `>`, but it is implied that it is there at the start of each line of code. The `[1]` indicates first line of output.

Assigning objects

Data in R is stored as objects which, when allocated, are located in the workspace. Objects can be assigned in several ways. The traditional R method uses `<-` i.e. `X <- 2+2` assigns the value of `2+2` to the object `X`, thus `X` will return 4. You can also use `=` to achieve the same ends.

Example: Assigning objects

```
>A <- 2*4
>A
[1] 8
>B = 2+3
>B
[1] 5
>C <- A - B
>C
[1] 3
```

It is a good idea to find an assignment method that suits you and stick with it for consistency's sake. Throughout this document I will be using `<-` to assign objects within the R environment. Also note that when we assign an object it can be seen in the RStudio workspace pane.

Working directories

We discussed working directories in the previous section, mostly in the context of RStudio. Here we will cover changing working directories using the console.[15]

Getting your current working directory

```
getwd()          # returns the current working directory
```

Note: if you are new to programming I'll touch briefly on commenting. In the above code you'll notice some green text after a hash (`#`), this is a comment. Comments are not read/executed by the program, and are a way for the author to make notes in the code (for themselves and others). So when you are following examples you don't need to enter comments, they won't contribute anything but extra typing. ☺

Setting a new working directory

```
setwd("C:/users/mbenton/home")
dir()                                  # returns a list of files in current working directory
```

Keep a careful eye on your directory structure. In R you use `/` instead of the `\` that is implemented in Windows. Alternatively you can use `\\` i.e. `"C:\\users\\mbenton\\home"`. I suggest picking one and sticking with it as this will avoid confusion at a later stage.[16]

Navigating the R workspace

The default R workspace is usually hidden to the user, and as such there are commands that provide this information.

List loaded objects

```
ls()             # will list all loaded/stored objects from workspace
objects()
```

When using RStudio however, objects and loaded variables are easily viewed in the workspace pane. Information is also provided about their type (dataframe, matrix, etc.) and size (i.e. 2x2).

[15] Creating directories from the console is covered in the first exercise.
[16] I find using the single `/` easier and more efficient, also if you've used Unix before you'll feel right at home.

Objects can be cleared or removed from the workspace either selectively or collectively with `rm()`.

Remove an object(s) from workspace

```
# example
rm(A)                  # removes object A from workspace
rm(B, C)               # removes all selected objects (B, C) from workspace
## use code below with extreme caution!
rm(list = ls())        # removes EVERY object from the workspace
```

Be careful with your use of `rm()` as this completely removes data from the R environment – **it is gone forever!**[17]

The Library and Packages

The default R library is located in the main R install directory (e.g. `"C:\R-2.14.0dev\library"`). With a fresh install this contains all the base packages included with R base release. You can check your default library path by using the `.libPaths()` function. When you use the `install.packages("package1")` function, "package1" will be installed in this directory. This package can then be loaded into the R environment using `library("package1")`.

```
.libPaths()                          # returns the path(s) of installed libraries
library()                            # display's all installed packages
install.packages("kinship2")         # will download and install selected package from CRAN
library("kinship2")                  # loads selected package into R environment
detach("package:kinship2")           # unloads selected package from R environment
```

It should be noted that when you change to a different version of R you will have to re-download all of your packages. This is due to R creating new libraries with each installation (some packages aren't backwards compatible). There are ways around this, but none that don't require a fair bit of blood sweat and tears... most of the time it's not worth the grief.[18]

Other useful commands

Some commands that we'll use in the first exercise. Many more can be found in the cheat sheet.

Quit

```
q()     # quits current R session, will ask for confirmationa and if you want to save
```

Garbage Collection

```
gc()    # releases system memory from R's clutches (only usefully when working with huge data
```

Additional commands to explore data in the workspace

```
object                 # prints the object on screen
length(object)         # number of elements or components
str(object)            # structure of an object
class(object)          # class or type of an object
names(object)          # names
rownames(object)       # names of rows
colnames(object)       # names of columns
c(object, object,…)            # combine objects into a vector
cbind(object, object,…)        # combine objects as columns
rbind(object, object,…)        # combine objects as rows
```

[17] Or until you reload/reassign the data. Also remember removing objects from workspace won't delete files.
[18] There is one method that involves editing your .Rprofile and setting lib paths, I'll cover this if people want.

Exercise 1: Taking the Plunge

This is the first of the 4 exercises, and as such is about getting used to working with the command line (or terminal) interface, as well as RStudio. For this exercise we will be using one of the inbuilt R example datasets, `mtcars`, to get an idea of how to navigate and view data in R. If you are interested in other example datasets you can type `data()` in your console to get a list of all that are available.

The first thing that we're going to do is create a new working directory. This is where example files will be stored, and where new files will be written to. **Remember to modify the directory path below to the correct place in your system!** One final note before getting underway, **don't be scared of ERROR messages!** At first they will be frustrating and unusual, but as you progress you'll begin to understand what's happing and how to fix it. You only start to worry when something's not working and you're NOT getting any ERROR output!

Creating a new working directory

```
## create a new working directory
dir.create("G:/Tutorials/R_tuts/Workshop_Oct5th/Rtut", recursive = TRUE, showWarnings = FALSE)

## some nifty code to create new dir with date in path (more advanced)
date <- Sys.Date()     # reports the current date and stores as object 'date'
outdir <- paste("G:/Tutorials/R_tuts/Workshop_Oct5th/Rtut", date, sep="_")
dir.create(outdir, recursive = TRUE, showWarnings = FALSE)          # create the directory

## now we set this as our working directory
setwd(outdir)          # change to the above newly created working directory
getwd()                # check that you're in the correct directory
```

Note: remember that you can always create your working directory from within Windows/Linux/Mac using the traditional methods and just point R/RStudio in the right direction. It is useful though learning to do it from the console as it saves time and gets you feeling more comfortable with using various commands.

Now that we have a new working directory it's time to explore! I'll get you started with a few basic functions/commands, if you're unsure of what a function is doing make sure to check out the help section of R – type either `?fun` or `help(fun)`, where `fun` is the function you want help with. If you're not sure of the exact function, try `search.help("thingtosearch")`.

Exploring data and other useful functions

```
# this example introduces data() - a collection of sample datasets
data()                 # opens a list in the top pane of all availabe datasets
mtcars                 # will display the mtcars data
class(mtcars)          # display class of object
str(mtcars)            # display the objects structure
length(mtcars)         # length of object
names(mtcars)          # object names of mtcars
colnames(mtcars)       # column names of mtcars
rownames(mtcars)       # row names of mtcars
head(mtcars)           # displays the first 5 rows of data

# now we're going to 'copy' the mtcars data to a new object
cars2 <- mtcars        # data sets can be assigned exactly as we did above
cars2                  # check to confirm that it's the same
str(cars2)
```

Have a bit of a 'play' around in the R console, use the commands we've covered to view and create objects. When everyone's ready we'll delve into the next section.[19] Before we do make sure you clear your workspace (now's the time for `rm(list = ls())`).

[19] Where things are really going to step up a notch!!

Section 2: Data Management in R

One of the most useful features of R is its ability to handle and manipulate data. Most of the time we find ourselves doing this in software such as Excel, but this just isn't practical with some datasets. Also the ability to create scripts to achieve a specific 'job' in R makes data management faster and more reliable. Section 2 introduces you to a selection of functions that will demonstrate the power of working with data in R.

Importing data into R

Usually you will obtain a dataframe by importing it from SAS, SPSS, Excel, Stata, a database, or an ASCII file. To create it interactively, you can do something like the following.

Example: Input dataframe using the keyboard

```
# create a simple dataframe from scratch
age <- c(25, 30, 56)
gender <- c("male", "female", "male")
weight <- c(160, 110, 220)
mydata <- data.frame(age, gender, weight)
```

Importing data from 'raw' text files

When I talk about 'raw' text files I mean formats such as; tab delimited (.txt), white spaced (.txt), and comma separated (.csv). These file formats are the ideal import/export formats for the bulk of the data you will be working with in R.

Import

```
## read tab delim file
read.table("beaver1.txt", sep="\t", head=TRUE, nrows=5)
## read tab delim file and store as object
beaver1.txt <- read.table("beaver1.txt", sep="\t", head=TRUE)
beaver1.txt    # see footnote[20]
## read .csv file
beaver1.csv <- read.table("beaver1.csv", head=T, sep=",")
# first row contains variable names (head=T), comma is separator (sep=",")
```

Note: R will accept `=TRUE` **or** `=T`**, as well as** `=FALSE` **or** `=F`**, for most arguments.**

Importing data from Excel files

It is my preference to import data into R straight from raw text files. In my opinion the best way to read an Excel file is to export it to a tab, white space, or comma delimited file and then import it to R using the method above. However it is possible to import data from Excel files. If you wish to do so you will require several additional packages: `"xlsReadWrite"` and `"xlsx"`.

Download and Install Required Packages

```
install.packages("xlsx")              # will download and install "xlsx"
install.packages("xlsReadWrite")      # will download and install "xlsReadWrite"
library("xlsx")                       # load "xlsx"
library ("xlsReadWrite")              # load "xlsReadWrite"
```

[20] **IMPORTANT: it is good practice to view each object you create. I have given an example in the above code using beaver1.txt. To save on code and typing I have not included this in every example, but you should get used to checking your data as you work.**

Import data from .xls file (Excel 2003 and earlier)

```
## import .xls (filetype before Excel 2007)
read.xls("beaver1.xls")
beaver1.xls <- read.xls("beaver1.xls")
head(beaver1.xls)
```

Note: apparently xlsReadWrite only works on 32 bit systems at this stage (as of 29/09/2011)

Import data from .xlsx file (Excel 2007/2010) **"read.xlsx" now compatible with .xls files**

```
## import .xlsx (filetype of Excel 2007/2010)
read.xlsx("beaver1.xlsx", 1)  # reads the first worksheet from chosen .xlsx file
beaver1.xlsx <- read.xlsx("beaver1.xlsx", 1)
head(beaver1.xlsx)
```

Exporting data from R

There are numerous methods for exporting R objects into other formats.

Export to a tab delimited text file

```
write.table(mtcars, "mtcars_out.txt", sep="\t", col.names=TRUE, row.names=TRUE, quote=FALSE)
```

Export to a comma separated text file

```
write.table(mtcars, "mtcars_out.csv", sep=",", col.names=TRUE, row.names=TRUE, quote=FALSE)
# note the seperator has changed (sep=",") as has the file extension (.csv)
```

To export to Excel formats, you will need the `"xlsReadWrite"` package for .xls files and the `"xlsx"` package for .xlsx files.

Export to an Excel file

```
# create .xls
write.xls(mtcars, "mtcars_out.xls")

# create/append .xlsx
write.xlsx(mtcars, "mtcars_out.xlsx")
# note: this can create a new Excel file, or append to an existing file
```

Viewing your Data

Once you've got your data into R the first thing you'll notice is the lack of traditional spread sheet like layout. So how do we go about viewing our data? We can make use of many of the functions we learnt about in the previous section; one of the most used is `head(mydata)` – where `mydata` is the dataframe of interest. As mentioned before, `head()` provides an overview of the first 5 rows of a dataframe, and includes all columns and column titles (if present).

If you want to view a specific column in a dataframe you can do so by using `$`, i.e. `mtcars$hp` will show the `hp` column of the `mtcars` dataframe. **Note:** Typing `mtcars$` and pressing tab will bring up a list of all columns and you can select the one that you want.

Example: Using $ to view columns

```
mtcars$hp      # shows hp column
mtcars$mpg     # shows mpg column
mtcars$        # press tab after $ to get a list of all columns
```

Example: Another way to look at columns and rows

```
mtcars[,1]              # shows column 1 of mtcars as a list of data points
mtcars[1]               # shows column 1 of mtcars with rownames
mtcars[1,]              # shows row 1 of mtcars
mtcars[1:5,1:4]         # shows rows 1-5 and columns 1-4 of mtcars
```

Attaching and detaching dataframes/objects

In the example above we learnt how to view columns in a dataframe using `$`. When you start writing more extensive R scripts you'll soon find that constantly calling specific columns becomes a bit of a chore. Luckily the `attach()` and `detach()` functions make life that much easier. Using `attach(mydata)`, the dataframe `mydata` is attached to the R search path. This means that the dataframe is searched by R when evaluating a variable, so objects (including columns) in the dataframe can be accessed by simply giving their names.

Example: using attach and detach

```
attach(mtcars)          # attaches the mtcars data to search path
hp                      # hp column
mpg                     # mpg column
detach(mtcars)          # detachs mtcars from search path
```

Sorting Data

To sort a dataframe in R, use the `order()` function. By default, sorting is ASCENDING. Prepend the sorting variable by a minus sign to indicate DESCENDING order. Here are some examples.

Example: Sorting using mtcars dataset

```
attach(mtcars)
# sort by mpg
newdata <- mtcars[order(mpg),]
# sort by mpg and cyl
newdata2 <- mtcars[order(mpg, cyl),]
#sort by mpg (ascending) and cyl (descending)
newdata3 <- mtcars[order(mpg, -cyl),]
detach(mtcars)
```

Now compare `mtcars$mpg`, `newdata$mpg`, `newdata2$mpg` and `newdat3$mpg` (hint: do this by typing each one separately and viewing it).

Subsetting Data

R has powerful indexing features for accessing object elements. These features can be used to select and exclude variables and observations. The following code snippets demonstrate ways to keep or delete variables and observations and to take random samples from a dataset.[21]

Selecting (Keeping) Variables

```
# select variables mpg, hp, gear
myvars <- c("mpg", "hp", "gear")
newdata <- mtcars[myvars]      # subsets based on selected variables, compare to mtcars.
# select 1st and 5th thru 10th variables, compare to mtcars.
newdata <- mtcars[c(1,5:10)]
```

[21] `subset` becomes a very useful tool when working with data in R – try and understand how it is working and what it can do.

Excluding (DROPPING) Variables

```
# exclude variables mpg, hp, gear
myvars <- names(mtcars) %in% c("mpg", "hp", "gear")
newdata <- mtcars[!myvars]
# exclude 3rd and 5th variable
newdata <- mtcars[c(-3,-5)]
# delete variables v3 and v5
mtcars$mpg <- mtcars$gear <- NULL
```

Selecting Observations

```
# first 5 observerations
newdata <- mtcars[1:5,]       # this selects rows 1:5
# based on variable values
newdata <- mtcars[ which(mtcars$gear==3 & mtcars$hp > 150), ]  # selects 3 gears & > 150 hp
# or
attach(mtcars)
newdata <- mtcars[ which(gear==3 & hp > 150),]       # same as above, but less typing
detach(mtcars)
```

Selection using the Subset Function

The `subset()` function is the easiest way to select variables and observations. In the following example, we select all rows that have a value of age greater than or equal to 20 or age less then 10. We keep the ID and Weight columns.

```
# using subset function
mtcars.A <- mtcars     # copy data to object
# set subset, here using "mpg" and >=21 and <15
mtcars.B <- subset(mtcars, mtcars$mpg >= 21 | mtcars$mpg < 15)
# select as above but only include columns "mpg", "hp", and "gear"
mtcars.C <- subset(mtcars, mtcars$mpg >= 21 | mtcars$mpg < 15, select=c("mpg", "hp","gear"))
# selects as above but columns 1:6
mtcars.D <- subset(mtcars, mtcars$mpg >= 21 | mtcars$mpg < 15, select=c(1:6))
```

Merging Data

Merging large datasets can sometimes be an issue (as well as onerous). R has several functions that quickly allow you to combine multiple sets of data into on larger set.

> **Note:** It is important to remember not to overwrite objects in the workspace unless you are sure they are not required (or you want to overwrite them).

Adding Columns

To merge two dataframes (datasets) horizontally, use the `merge()` function. In most cases, you join two dataframes by one or more common key variables – for me this is usually an "id" column.

```
# can use cbind()
cars.cbind <- cbind(mtcars.C, mtcars.D)       # what's wrong with this 'picture'?
# merge two dataframes by selected variables
cars.merge <- merge(mtcars.C, mtcars.D, by=c("mpg", "hp"), sort=FALSE)
# below is better, why? Can you tell what I did? Hint: compare cars.merge and cars.merge2
mtcars.C <- cbind(rownames(mtcars.C), mtcars.C)
colnames(mtcars.C)[1] <- "make"
mtcars.D <- cbind(rownames(mtcars.D), mtcars.D)
colnames(mtcars.D)[1] <- "make"
cars.merge2 <- merge(mtcars.C, mtcars.D, by=c("make", "mpg", "hp"), sort=FALSE)
```

Adding Rows

To join two dataframes (datasets) vertically, use the `rbind()` function. The two dataframes must have the same variables, but they do not have to be in the same order – i.e. use this function when you have additional individuals to add to a dataset.

```
cardata1 <- mtcars[1:5,]                    # dataframeA: selected rows 1:5 of mtcars
cardata2 <- mtcars[10:20,]                  # dataframeB: selected rows 10:20 of mtcars
cardata.rbind <- rbind(cardata1, cardata2)  # this is the final dataset, combined A B by row
```

If dataframeA has variables that dataframeB does not, then either: 1) Delete the extra variables in dataframeA or 2) Create the additional variables in dataframeB and set them to NA (missing) before joining them with rbind.

Random Samples

Use the `sample()` function to take a random sample of size n from a dataset.

```
# take a random sample of size 15 from a mtcars dataset
# sample without replacement
mysample <- mtcars[sample(1:nrow(mtcars), 15, replace=FALSE),]
```

Reshaping Data

Transpose

Use the `t()` function to transpose a matrix or a dataframe. In the latter case, rownames become variable (column) names. **Be very careful with this function, it sometimes produces unexpected results!**

```
# example using built-in dataset
mtcars
t(mtcars)
```

Data Type conversion

Type conversions in R work as you would expect. For example, adding a character string to a numeric vector converts all the elements in the vector to character. This might sound odd, let me explain: R categorises your data into a type (characters, matrix… etc.), so sometimes when you're getting an error or weird result it could be due to an 'incorrect' data type. You can change as below.

Use is.foo to test for data type foo. Returns TRUE or FALSE

Use as.foo to explicitly convert it.

```
is.numeric(),is.character(),is.vector(),is.matrix(),is.data.frame()

as.numeric(),as.character(),as.vector(),as.matrix(),as.data.frame()
```

Dealing with NA's

In R, missing values are represented by the symbol NA (not available). Impossible values (e.g., dividing by zero) are represented by the symbol NaN (not a number). Unlike SAS, R uses the same symbol for character and numeric data.

Testing for Missing Values

```
is.na(x)              # returns TRUE of x is missing
y <- c(1,2,3,NA)
is.na(y)              # returns a vector (F F F T)
```

Recoding Values to Missing

```
# use mtcars.A to recode 3 gears to missing for variable "gear"
mtcars.A[mtcars.A$gear==3,"gear"] <- NA
mtcars.A              # see what it looks like
summary(mtcars.A)     # does summary display the NAs?
is.na(mtcars.A)       # another way of looking at it
```

Excluding Missing Values from Analyses

Arithmetic functions on missing values yield missing values.

```
x <- c(1,2,NA,3)
mean(x)                 # returns NA
mean(x, na.rm=TRUE)     # returns 2
```

The function `complete.cases()` returns a logical vector indicating which cases are complete.

```
# list rows of data that have missing values
mtcars.A[!complete.cases(mtcars.A),]
```

The function `na.omit()` returns the object with listwise deletion of missing values. [22]

```
# create new dataset without missing data
cars.nonmissing <- na.omit(mtcars.A)
```

Looking for Unique Values

When working with large datasets it is important to

`identical()` - The safe and reliable way to test two objects for being exactly equal. It returns TRUE in this case, FALSE in every other case.

`unique()` - returns a vector, data frame or array like x but with duplicate elements/rows removed. It is important to note that `unique()` picks the first occurrence it comes across, this behaviour can be modified

Example:

```
unique(iris$Sepal.Length)
iris.sub <- unique(iris$Sepal.Length)
```

`duplicated()` - Determines which elements of a vector or data frame are duplicates of elements with smaller subscripts, and returns a logical vector indicating which elements (rows) are duplicates.
`!duplicated()` - looks for values that aren't repeated

Example: using duplicated/!duplicated to subset

```
iris.test <- !duplicated(iris)
# subset iris based on 'unique' Speal.Length
iris.test <- subset(iris, !duplicated(iris$Sepal.Length))
iris.test

mtcars.test <- !duplicated(mtcars$hp)
# subset mtcars based on 'unique' hp
mtcars.test <- subset(mtcars, !duplicated(mtcars$hp))
mtcars.test
```

[22] You really have to think if this is what you want to do before using `na.omit`, as this removes all cases that contain NAs (you could be losing a lot of data). You might be better off excluding them at the analysis level.

Exercise 2: Using NIHS and NIES data as an example

This is the second exercise of the workshop. Here we start to explore several methods of data management in R, including; QC, import/export, and merging. The 4 files that you require for this exercise are located in file `Workshop_NI_exampledata.zip`, they are: `NI_IDsheet.xlsx`, `NIHS2009_pheno.xlsx`, `NIHS2000_pheno.xlsx` and `NIES2008_pheno.xlsx`.

Overview of this exercise:

1. Good QC practice
2. Export tab delimited (.txt) files from Excel for sample NIHS and NIES data.
3. Load both of these files into R workspace (NIHS.pheno & NIES.pheno).
4. Explore this data briefly using summary.
5. Check for missing data.
6. Subset/filter based on age =>18 years of age for each dataset.
7. Merge data based on select variables (specific ids and gender).
8. Explore sorting/ordering based on given variables.
9. Export this file(s) as tab delimited and then check by opening in Excel.

1. A note on good Quality Control (QC) practices

Before we launch into the exercise it is a good time to mention several steps that allow you to quickly and easily 'check' your data. This is a nice way to get 'peace of mind' that everything is as it should be and nothing untoward has happened to your data. The functions used are ones that we have covered above, so should look somewhat familiar to you.

- First it is important to check the head and tail of you data. Using the `head()` and `tail()` functions allows us to view the first 5 and last 5 entries to make sure the data is in the correct 'format' and that the complete data set has loaded into R's workspace.
- Viewing the structure of your data using `str()` allows you to see how many observations (rows) and variables (columns) have been loaded in.
- Using `summary()` you can get an idea of the 'spread' of your data, as well as the number of missing entries (NAs) in each column.
- Another useful function is `identical()`. If you have 2 versions of the same file and are wondering if they are different at all load them into R and use:

```
identical(myfileA, myfileB)    # where myfileA and myfileB are the two files for comparison
```

If the files are identical you'll get `TRUE` as an output, if you get `FALSE` you'll know that they differ and then you can use the above means to identify exactly where and how they differ.

2. Exporting/saving as 'raw text' from Excel

I'm assuming that most people know how to export an Excel worksheet as a raw text file. If you're not 100% sure here is a brief explanation.

- In Excel, go to the 'File | Save As…' menu
- Change the 'Save as type:' option in the drop down list to "Text (Tab delimited)(*.txt)"
- Enter the file name you wish to save as and select 'save'
- Done.

Note: you can use the exact same method to save as .csv or white space separated (.txt), as well as numerous other file types. You can even save as a PDF from this menu in Excel 2007/2010.

3. Load files into R (and some QC steps…)

```
## load in the complete list of NI ID's
NI.IDs <- read.table("NI_IDsheet.txt", sep="\t", head=T)
head(NI.IDs)           # first 5 entries
tail(NI.IDs)           # last 5 entries
str(NI.IDs)            # check the structure, how many obs. & vars? Is this the same as Excel?

## load in 2009 NIHS path data
NIHS2009.pheno <- read.table("NIHS2009_pheno.txt", sep="\t", head=T)
head(NIHS2009.pheno)   # first 5 entries
tail(NIHS2009.pheno)   # last 5 entries
str(NIHS2009.pheno)    # check the structure, how many obs. & vars? Is this the same as Excel?
summary(NIHS2009.pheno)        # summary data for all
attach(NIHS2009.pheno) # attach NIHS pheno
summary(BMI)           # summary of just BMI
summary(AGE)           # summary of just AGE
detach(NIHS2009.pheno) # detach NIHS pheno

## load in 2000 NIHS path data
NIHS2000.pheno <- read.table("NIHS2000_pheno.txt", sep="\t", head=T)
head(NIHS2000.pheno)           # first 5 entries
tail(NIHS2000.pheno)           # last 5 entries
str(NIHS2000.pheno)            # check the structure, how many obs. & vars? Same as Excel?
summary(NIHS2000.pheno)        # summary data for all

## load in 2008 NIES pheno data
NIES.pheno <- read.table("NIHS2008_pheno.txt", sep="\t", head=T)
head(NIES.pheno)       # first 5 entries
tail(NIES.pheno)       # last 5 entries
str(str(NI_IDs))       # check the structure, how many obs. & vars? Is this the same as Excel?
summary(NIES.pheno)    # summary data for all
```

Note: with summary you'll see a breakdown of the basic stats for each variable. These are calculated with NAs excluded – you can also see the number of NAs for each variable at the bottom of each summary column.

4. Subset based on age

Here we are going to subset each dataset based on the age variable. We want to select individuals that are 18 years or older. Here's one way to do this:

```
## Subset based on age
# 2000 NIHS
NIHS2000.pheno <- subset(NIHS2000.pheno, NIHS2000.pheno$AGE_2000 >= 18)
# 2009 NIHS
NIHS2009.pheno <- subset(NIHS2009.pheno, NIHS2009.pheno$AGE >= 18)
```

5. Generate summary's for the data

As in the example above, this is achieved with the `summary()` function. I'll list a few uses of `summary()`:

```
## Generate summary's for the data
# basic summary use
summary(NIHS2000.pheno)           # total summary
summary(NIHS2000.pheno$BF_2000)   # summary of Body Fat only

# 2000 NIHS
NIHS2000.summary <- summary(NIHS2000.pheno)
NIHS2000.summary

# 2009 NIHS
NIHS2009.summary <- summary(NIHS2009.pheno)
NIHS2000.summary

# NIES
NIES.summary <- summary(NIES.pheno)
NIES.summary
## these summary tables can be written out to text files for later use (reports)
## this is covered at the end of the exercise
```

6. Check for missing data

You would have noted that in our uses of `summary()` above the number of NA's are being reported for each variable. Here we'll cover a few different methods of checking for missing values.

```
## Check for missing data
# as we've seen summary() reports the number of missing (NA) values for each variable
summary(NIHS2009.pheno)

# we can also use is.na(), which reports total number of NA's in a dataset
is.na(NIHS2009.pheno)          # notice the printed readout, we don't want this so...
table(is.na(NIHS2009.pheno))   # reports total non-missing and missing (missing is TRUE)

# can use complete.cases to remove all individuals with missing data
NIHS2009.clean <- NIHS2009.pheno[complete.cases(NIHS2009.pheno),]
# you can also generate a list of individuals with missing data
NIHS2009.missing <- NIHS2009.pheno[!complete.cases(NIHS2009.pheno),]
# can also use na.omit to remove individuals with missing data
NIHS2009.clean2 <- na.omit(NIHS2009.pheno)
## Note: before removing NA's it's important to consider the amount of data you'll lose
```

7. Merging multiple datasets

Here we are going to merge all the NI phenotype data into one dataset. This first section of code looks at creating a dataset containing only those individuals that are enrolled in the 3 NI studies: 2000 NIHS, 2009 NIHS and NIES.

```
## create a merged file of individuals in all 3 studies
# first I would merge 2009 NIHS with the ID data
NI.merge <- merge(NI.IDs, NIHS2009.pheno, by="PID")

# then bring in the 2000 NIHS data
NI.merge <- merge(NI.merge, NIHS2000.pheno, by=c("PID", "LAB_ID"))
# we'll create a seperate object for NIHS crossover
NIHS.merge <- NI.merge

# finally we'll tie in the NIES data
NI.merge <- merge(NI.merge, NIES.pheno, by=c("NIES", "Gender"))  # merged by NIES ID!!

head(NI.merge)  # check final merged file
# we now have a total of 256 individuals across all 3 studies
```

This next section of code demonstrates how to create a file of ALL NI individuals, and will set those that don't have phenotype data to missing for those values.

```
## create a merged file of ALL individuals
# first I would merge 2009 NIHS with the ID data
NI.merge.all <- merge(NI.IDs, NIHS2009.pheno, by="PID", all.x=TRUE)

# then bring in the 2000 NIHS data
NI.merge.all <- merge(NI.merge.all, NIHS2000.pheno, by=c("PID", "LAB_ID"), all.x=TRUE)

# finally we'll tie in the NIES data
NI.merge.all <- merge(NI.merge.all, NIES.pheno, by=c("NIES", "Gender"), all.x=TRUE)

head(NI.merge.all)  # check final merged file
```

8. Sorting/ordering on variables

Here we are going to sort our dataframe(s) based on specific variables, i.e. ID, Age, DOB.

```
## Sorting/ordering using AGE
NIHS.merge.ordered <- NIHS.merge[order(NIHS.merge$AGE),]     # ascending
head(NIHS.merge.ordered)
tail(NIHS.merge.ordered)
NIHS.merge.ordered <- NIHS.merge[order(-NIHS.merge$AGE),]    # descending
head(NIHS.merge.ordered)
tail(NIHS.merge.ordered)
```

We can do the same for LAB_ID:

```
## sort by LAB_ID
NIHS.merge.ordered <- NIHS.merge[order(NIHS.merge$LAB_ID),]
head(NIHS.merge.ordered)
tail(NIHS.merge.ordered)
```

You can also apply multiple sorting variables, i.e. let's try body fat and SBP.

```
## sort by high to low BF and SBP
NIHS.merge.ordered <- NIHS.merge[order(-NIHS.merge$Body.Fat., -NIHS.merge$Adj_SBP),]
# note the use of - above to indicate we want to sort from high to low
head(NIHS.merge.ordered)
tail(NIHS.merge.ordered)
```

9. Export/Save newly created data out to files

Now that we have several newly subsetted, merged, and order datasets it's time to write these out as files that can be shared, stored, or loaded into other software.

Write out as tab delimited files (.txt):

```
## Export/Save newly created data out to files
# write tab delim
write.table(NIHS.merge, "NIHS.longitudinal.txt", sep="\t", col.names=T, row.names=F, quote=F)
write.table(NI.merge, "NI.merged.txt", sep="\t", col.names=T, row.names=F, quote=F)
```

Write out as comma separated files (.csv):

```
# write .csv
write.table(NIHS.merge, "NIHS.longitudinal.csv", sep=",", col.names=T, row.names=F, quote=F)
write.table(NI.merge, "NI.merged.csv", sep=",", col.names=T, row.names=F, quote=F)
```

Write out as Excel files (.xls and .xlsx):

```
# write Excel files
# .xlsx
write.xlsx(NIHS.merge, "NIHS.longitudinal.xlsx", sheetName="NIHS", col.names=T, row.names=F)
# .xls
write.xlsx(NIHS.merge, "NIHS.longitudinal.xls", sheetName="NIHS", col.names=T, row.names=F)
```

Once these files have been written out we can clean up the workspace. Enter `rm(list = ls())` at the console, this removes all objects from the workspace – we can now start a fresh in Section 3.

This officially ends Exercise 2, and Section 2. Congratulations for getting through. Now before we move to the next section I want to mention a bit more about source (script) files. These will usually have an extension .r, i.e. `myscript.r`.

You may have noticed one of these files in the directory with the NI phenotype data, `Workshop_Exercise2_source.r`. This file contains all of the script from Exercise 2, as well as additional comments. Feel free to open this up in RStudio by using 'File | Open File…' and pointing to the script file. If you wish to run through Exercise 2 at a later stage everything you need should be included in this file. Feel free to modify/tweak this script to your heart's content – this is the best way to learn more about R and scripting.[23]

[23] You can run code line by line, or in chunks from the source tab in RStudio. Select the line you would like to run and then either; press the 'run line' button, or use [ctrl]+[Enter].

Section 3: Basic Statistical methods

R is primary a statistical language - over the years it has morphed into much more – but at its heart there is a powerful engine for advanced statistical modelling and analysis. I haven't yet explored much in the way of advanced statistical analysis in R, and much of what I have looked at isn't really appropriate for this workshop.[24] In light of this, this section aims to provide a base understanding of some of the statistical functions of R. Hopefully you'll be able to branch out and explore using R's help system and the internet to find more statistical tests to fit your needs.[25]

The bulk of these examples have been modified from those found at Quick-R (see references at the start of the manual for url).

Exploring data – basic statistical commands

R provides a wide range of functions for obtaining summary statistics. One method of obtaining descriptive statistics is to use the `sapply()` function with a specified summary statistic.

```
# get means for variables in dataframe mydata
# excluding missing values
sapply(mydata, mean, na.rm=TRUE)      # na,rm=TRUE excludes missing values from reported stats
```

Possible parameters used in `sapply` include mean, sd, var, min, max, med, range, and quantile:

```
# test sapply in a few datasets using various functions
# mtcars
sapply(mtcars, mean, na.rm=TRUE)
sapply(mtcars, sd, na.rm=TRUE)
sapply(mtcars, var, na.rm=TRUE)
# iris
sapply(iris, min, na.rm=TRUE)
sapply(iris, max, na.rm=TRUE)
sapply(iris, med, na.rm=TRUE)
# DNase
sapply(DNase[,2:3], range, na.rm=TRUE)
sapply(DNase[,2:3], quantile, na.rm=TRUE)
```

There are also several R functions designed to provide a range of descriptive statistics at once. For example:

```
# mean, median, 25th and 75th quartiles, min, max
summary(iris)  # obviously we're already very familiar with summary (useful tool!)

# Tukey min, lower-hinge, median, upper-hinge, max
fivenum(x)     # only works on 1 variable at a time (x)
# example using data from DNase
fivenum(DNase$density)
```

Frequencies and Crosstabs

This section describes the creation of frequency and contingency tables from categorical variables, along with tests of independence, measures of association, and methods for graphically displaying results.

[24] If anyone is interested in Principle Components Analysis (PCA) I could put together a small manual with R scripts from some of the work that I've done.

[25] As far as I can see there are inbuilt functions and external packages that allow R to offer all the features (and more) of software such as SPSS and Graphpad Prism – so if you use these often you could try running you're analysis in R.

Generating Frequency Tables

R provides many methods for creating frequency and contingency tables. Three are described below. In the following examples, assume that A, B, and C represent categorical variables.

table

You can generate frequency tables using the `table()` function, tables of proportions using the `prop.table()` function, and marginal frequencies using `margin.table()`.

```
# 2-Way Frequency Table
attach(mtcars)
mytable <- table(cyl, gear)    # cyl will be rows, gear will be columns
mytable                        # print table

margin.table(mytable, 1)       # cyl frequencies (summed over gear)
margin.table(mytable, 2)       # cyl frequencies (summed over gear)

prop.table(mytable)            # cell percentages
prop.table(mytable, 1)         # row percentages
prop.table(mytable, 2)         # column percentages
detach(mtcars)
```

`table()` can also generate multidimensional tables based on 3 or more categorical variables. In this case, use the `ftable()` function to print the results more attractively.

```
# 3-Way Frequency Table
mytable <- table(cyl, gear, carb)
ftable(mytable)
```

Note: Table ignores missing values. To include NA as a category in counts, include the table option `exclude=NULL` if the variable is a vector. If the variable is a factor you have to create a new factor using `newfactor <- factor(oldfactor, exclude=NULL)`.

xtabs

The `xtabs()` function allows you to create cross tabulations using formula style input.

```
# 3-Way Frequency Table
mytable <- xtabs(~A+B+c, data=mydata)
ftable(mytable) # print table
summary(mytable) # chi-square test of indepedence

# test using mtcars
# 3-Way Frequency Table
mytable <- xtabs(~cyl+gears+carb, data=mtcars)
ftable(mytable) # print table
summary(mytable) # chi-square test of indepedence
```

If a variable is included on the left side of the formula, it is assumed to be a vector of frequencies (useful if the data have already been tabulated).

Crosstable

The `CrossTable()` function in the `gmodels` package produces cross tabulations modelled after PROC FREQ in SAS or CROSSTABS in SPSS. It has a wealth of options.

```
# 2-Way Cross Tabulation
install.packages("gmodels")
library(gmodels)
CrossTable(mtcars$gear, mtcars$carb)
```

There are options to report percentages (row, column, cell), specify decimal places, produce Chi-square, Fisher, and McNemar tests of independence, report expected and residual values (Pearson, standardized, adjusted standardized), include missing values as valid, annotate with row and column titles, and format as SAS or SPSS style output! (See `help(CrossTable)` for more details)

Tests of Independence

Chi-Square Test

For 2-way tables you can use `chisq.test(mytable)` to test independence of the row and column variable. By default, the p-value is calculated from the asymptotic chi-squared distribution of the test statistic. Optionally, the p-value can be derived via Monte Carlo simulation.

```
mytable <- table(mtcars$carb, mtcars$cyl)
chisq.test(mytable)
```

Fisher Exact Test

`fisher.test(x)` provides an exact test of independence. `x` is a two dimensional contingency table in matrix form.

Mantel-Haenszel test

Use the `mantelhaen.test(x)` function to perform a Cochran-Mantel-Haenszel chi-squared test of the null hypothesis that two nominal variables are conditionally independent in each stratum, assuming that there is no three-way interaction. `x` is a 3 dimensional contingency table, where the last dimension refers to the strata.

Loglinear Models

You can use the `loglm()` function in the MASS package to produce log-linear models. For example, let's assume we have a 3-way contingency table based on variables A, B, and C.

```
install.packages("MASS")
library(MASS)
mytable <- xtabs(~A+B+C, data=mydata)
```

We can perform the following tests:

Mutual Independence: A, B, and C are pairwise independent.

```
 loglm(~A+B+C, mytable)
```

Partial Independence: A is partially independent of B and C (i.e., A is independent of the composite variable BC).

```
loglin(~A+B+C+B*C, mytable)
```

Conditional Independence: A is independent of B, given C.

```
loglm(~A+B+C+A*C+B*C, mytable)  # No Three-Way Interaction

loglm(~A+B+C+A*B+A*C+B*C, mytable)
```

t-tests

The `t.test()` function produces a variety of t-tests. Unlike most statistical packages, the default assumes unequal variance and applies the Welsh df modification.

```
# independent 2-group t-test
t.test(y~x)                     # where y is numeric and x is a binary factor
t.test(mtcars$wt~ mtcars$am)

# independent 2-group t-test
t.test(y1, y2)                          # where y1 and y2 are numeric
t.test(mtcars$mpg, mtcars$hp)           # example

# paired t-test
t.test(y1, y2, paired=TRUE)    # where y1 & y2 are numeric
t.test(mtcars$mpg, mtcars$hp, paired=TRUE)    # example

# one sample t-test
t.test(y, mu=3)                # Ho: mu=3
```

You can use the `var.equal = TRUE` option to specify equal variances and a pooled variance estimate. You can use the `alternative="less"` or `alternative="greater"` option to specify a one tailed test.

Exploring Correlations

You can use the `cor()` function to produce correlations and the `cov()` function to produces covariance's.

A simplified format is `cor(x, use=, method= )` where:

- `x` matrix or dataframe
- `use` specifies the handling of missing data. Options are `all.obs` (assumes no missing data - missing data will produce an error), `complete.obs` (listwise deletion), and `pairwise.complete.obs` (pairwise deletion)
- `method` specifies the type of correlation. Options are `pearson`, `spearman` or `kendall`.

```
# Correlations/covariances among numeric variables in
# dataframe mtcars. Use listwise deletion of missing data.
cor(mtcars, use="complete.obs", method="kendall")
cov(mtcars, use="complete.obs")
```

Unfortunately, neither `cor()` or `cov()` produce tests of significance, although you can use the `cor.test()` function to test a single correlation coefficient.

The `rcorr()` function in the Hmisc package produces correlations/covariance's and significance levels for Pearson and Spearman correlations. However, input must be a matrix and pairwise deletion is used.

```
# Correlations with significance levels
install.packages("Hmisc")
library(Hmisc)
rcorr(x, type="pearson") # type can be pearson or spearman
#mtcars is a dataframe
rcorr(as.matrix(mtcars))
```

You can use the format `cor(X, Y)` or `rcorr(X, Y)` to generate correlations between the columns of X and the columns of Y. This is similar to the VAR and WITH commands in SAS PROC CORR.

```
# Correlation matrix from mtcars with mpg, cyl, and disp as rows and hp, drat,
# and wt as columns
x <- mtcars[1:3]
y <- mtcars[4:6]
cor(x, y)
```

Regression Modelling

The amount of regression options available in R and associated packages is massive. I have included a few of the worked examples from R help below:

```
## several examples from R help
require(graphics)

## Annette Dobson (1990) "An Introduction to Generalized Linear Models".
## Page 9: Plant Weight Data.
ctl <- c(4.17,5.58,5.18,6.11,4.50,4.61,5.17,4.53,5.33,5.14)
trt <- c(4.81,4.17,4.41,3.59,5.87,3.83,6.03,4.89,4.32,4.69)
group <- gl(2,10,20, labels=c("Ctl","Trt"))
weight <- c(ctl, trt)
lm.D9 <- lm(weight ~ group)
lm.D90 <- lm(weight ~ group - 1) # omitting intercept

anova(lm.D9)
summary(lm.D90)

opar <- par(mfrow = c(2,2), oma = c(0, 0, 1.1, 0))
plot(lm.D9, las = 1)      # Residuals, Fitted, ...
par(opar)

## Dobson (1990) Page 93: Randomized Controlled Trial :
counts <- c(18,17,15,20,10,20,25,13,12)
outcome <- gl(3,1,9)
treatment <- gl(3,3)
print(d.AD <- data.frame(treatment, outcome, counts))
glm.D93 <- glm(counts ~ outcome + treatment, family=poisson())
anova(glm.D93)
summary(glm.D93)

## A Gamma example, from McCullagh & Nelder (1989, pp. 300-2)
clotting <- data.frame(
    u = c(5,10,15,20,30,40,60,80,100),
    lot1 = c(118,58,42,35,27,25,21,19,18),
    lot2 = c(69,35,26,21,18,16,13,12,12))
summary(glm(lot1 ~ log(u), data=clotting, family=Gamma))
summary(glm(lot2 ~ log(u), data=clotting, family=Gamma))
```

Further Possibilities

There are nearly limitless possibilities for statistical analysis in R; we've merely scratched the surface. I'll leave it up for you to explore the many additional packages that can be installed which offer powerful statistical tools. If you know the type of test/model you want to apply, a quick search of R help, CRAN, and the internet will see you on your way to achieving this in R. Remember, once you have run an analysis once and written the script, it is as simple as tweaking this for future analysis. The initial hurdle at the start is well worth the end result.

R for biomedical researchers – A worked example

For this example we will be using a subset of data from the 2010NIHS, mainly concentrating on BMI and it's interactions in the NI population. We will work through this example in a manner that is similar to exploratory statistics in a typical research setting; starting with descriptive statistics, moving to correlations, and then exploring further standard tests (chi-squared, t test). Plotting of this data will also briefly be touched on, with more advanced graphing/plotting options explored later in section 4.

If you would like to later replicate what we are doing here I have included all the R script for this exercise in the file `RWorkshop.Section3.r` (*hint:* this file can be loaded into the source window of RStudio so you can modify and run the script).

Loading the data

First the data for this session must be loaded as an object into R. Make sure you've set the correct working directory in RStudio – this directory must contain the files that we will be working with (`NIHS.2009.data.csv` in particular).[26]

```
## load NIHS.data
NIHS.data <- read.table("NIHS.2009.data.csv", head=T, sep=",")
```

There is another way to load files into R, allowing you to choose the file from a traditional 'explorer' window. The R function for this is `file.choose()`, feel free to play with this function (don't forget you can explore how it works with R help - `?file.choose()`

```
# or if you prefer selecting your file from explorer view:
NIHS.data <- read.table(file.choose(), head=T, sep=",")
```

Descriptive Stats

As with the start of any statistical analysis we're going to investigate the descriptive stats. As previous mentioned, our focus will be the BMI variable in this NIHS dataset. Before we begin we will quickly check the overall quality and consistency of the data:

```
# use summary() to get an overview of some basic stats
summary(NIHS.data)
```

One thing that I notice straight away is that the minimum age is less than 18 years. We do not want to include individuals under the age of 18 in further analysis, thus we need to filter the data to exclude these participants:

```
# check minimum age in the study group
summary(NIHS.data$AGE)
# filter for age >=18
NIHS.data <- subset(NIHS.data, NIHS.data$AGE>=18)
# now recheck the minimum age
summary(NIHS.data$AGE)
```

[26] **Tip:** Remember you can check your working directory with `getwd()`, and can list the files in your current working directory with `dir()`.

Right, we've filtered the data, removing any underage participants. Next thing is to generate some summary stats for BMI:

```
# just BMI
summary(NIHS.data$BMI)
```

From the above we're getting some basic descriptive stats: minimum, maximum, 1st quartile, 3rd quartile, and the mean. *What is the mean BMI for this current data set?*

We can further visualise this data with some basic plotting commands. First we'll use the boxplot:

```
# lets plot this to further explore:
boxplot(NIHS.data$BMI)
```

Now try the histogram (as well as the distribution plot):

```
# histogram:
hist(NIHS.data$BMI, breaks=50)
plot(density(NIHS.data$BMI, na.rm=T))
```

From these plots it should be fairly clear that there is a major outlier (BMI=82.10), that is far outside the normally distributed data. We will filter this out:

```
# filter for BMI <=60 (filtering by BMI less than or equal to 60)
NIHS.data <- subset(NIHS.data, NIHS.data$BMI<=60)

# now re-plot and double check it looks sensible:
boxplot(NIHS.data$BMI)
hist(NIHS.data$BMI, breaks=50)
plot(density(NIHS.data$BMI, na.rm=T))
```

Hopefully the distribution is looking a bit better. It should be noted that this is an initial and crude form of quality analysis and data cleaning. There are additional R packages available that deal with outlier pruning/filtering and many other data QC functions. *What is the mean BMI for this 'cleaned' data?*

Let's say we want to fit a distribution curve to our histogram, we can do that with a few extra lines of code. In this case it's a normal distribution and we would like to see how our data fits:

```
attach(NIHS.data)  # attach the data

# plotting BMI histogram with distribution normal curve
h <- hist(BMI, breaks=20, col="cadetblue", xlab="BMI", main="BMI Histogram with Normal Curve")
# the following code sets up and plots the normal curve:
xfit <- seq(min(BMI, na.rm=T), max(BMI, na.rm=T), length=40)
yfit <- dnorm(xfit, mean=mean(BMI, na.rm=T), sd=sd(BMI, na.rm=T))
yfit <- yfit*diff(h$mids[1:2])*length(BMI)
lines(xfit, yfit, col="darkblue", lwd=4)
# don't worry about this seeming complex, we'll cover later

detach(NIHS.data)  # detach the data
```

In the above code I've attached the `NIHS.data` object to somewhat cut down on typing (also makes it a bit easier to follow when you're starting out). You'll see a line of code to plot the histogram, while the rest of the code examines the distribution of the data and determines the parameters for the curve. You'll be able to adapt this code to other variables (this is something you can explore in the exercise at the end of this section). We'll cover exporting these beautiful graphs in section 4.

Correlations

Now we're going to perform exploratory correlation analysis comparing BMI with the other variables in the NIHS dataset. Using the `cor()` function we can generate a basic correlation matrix:

```
cor(NIHS.data, use="complete.obs", method="pearson") # you will notice an error....
```

After running the above you should be returned with an error. Don't worry; this is caused by the presence of a variable(s) that is non-integer/non-numeric. We can check on this by using our old friend `str()`:

```
# so check the str() of NIHS.data
str(NIHS.data)  # see that Gender is a factor, won't work with cor()
```

You'll notice that the Gender variable is indeed a factor, thus this is causing the error. Luckily we can use some prior R knowledge to select the variables that we want to run the correlation analysis on:

```
# try...
cor(NIHS.data[3:28], use="complete.obs", method="pearson") # don't need id so start at SEX (3)
```

The numbers in the square brackets indicate that we want to include variables 3 through 28, so we are ignoring variables 1 and 2 (they are not needed). The output from the above should be a correlation matrix; depending on your R output options this might look quite messy…

However, we can store this analysis as a matrix object that we can manipulate and write out to a file if we so choose.

```
## create an object of the matrix
NIHS.cor.matrix <- cor(NIHS.data[3:28], use="complete.obs", method="pearson")
# just display BMI
NIHS.cor.matrix[,"BMI"]
# write to file (can view in excel for example)
write.table(NIHS.cor.matrix, "NIHS.cor.matrix.csv", sep=",", col.names=T, row.names=T)
```

So now we have a matrix object, we can see all variables correlation to BMI, and have written this out to a `.csv` file. You might be wondering where the p-values are for these correlations. Well that's the drawback of the inbuilt chi squared function. However we have other options; such as the `Hmisc` package.

```
## can try for something more advanced/detailed
install.packages("Hmisc")     # run if you don't have Hmisc installed in you library
require("Hmisc")
# data needs to be a matrix with no characters/factors
NIHS.data.matrix <- as.matrix(NIHS.data[3:28])
rcorr(NIHS.data.matrix, type="pearson")
# or can do like this...
NIHS.cor.list <- rcorr(as.matrix(NIHS.data[3:28]), type="pearson")
# similar to above
NIHS.cor.list$r[,"BMI"]
NIHS.cor.list$n[,"BMI"]
NIHS.cor.list$P[,"BMI"]
```

The `Hmisc` package gives much more detail in terms of significance and number of samples included in the test; includes correlation stat, p-value and n. Again if you'd like to visualise these in spread sheet format please feel free to write them out to a file.

We can further explore correlations with BMI by creating scatter plots, this gives a general overview of any correlations, there strength, and direction. This is extremely straight forward and only requires you follow the formula: `plot(X,Y)` – where X=BMI and Y=trait of interest. Example:

```
## remember that we can plot these relationships quickly as a form of visualisation
plot(NIHS.data$BMI, NIHS.data$WEIGHT)
plot(NIHS.data$BMI, NIHS.data$HIP)
plot(NIHS.data$BMI, NIHS.data$CHOL_HDL_ratio)
```

NOTE: remember that data can be attached in R to make life much easier, i.e:

```
attach(NIHS.data)
plot(BMI, SBP)
plot(BMI, HIP)
detach(NIHS.data) # don't forget to detach when finished
```

There are many ways you can use scatter plots to portray correlations. Another popular method is to create a basic scatter plot matrix, useful for showing multiple correlations in one graph.

```
# basic scatter plot matrix
pairs(~BMI+WEIGHT+BF+SBP+WC, data=NIHS.data, main="Simple Scatterplot Matrix")
```

The number of variables can be increased to include as many as you want (within reason; you're screen size may limit what you'll actually see if you have LOTS of variables!!).

Hopefully you'll start to appreciate how quickly and easily you can explore your data in R. Overall the `plot()` function gives a nice graphical visualisation of the data, but like the inbuilt correlation function there is still a lot of information missing (R2, p-value). This is where external packages come in useful.

Performance Analytics: The all-in-one package for initial statistical analysis

I mentioned above that there was a package that had taken the basic idea behind the simple scatter plot matrix and improved it; that package is called `PerformanceAnalytics`. Instead of explaining what this package offers I'll leave it for you to explore by following the code below:

```
#### A very nice package for 'batch'/meta analyses of data
## setting up for some flash output
install.packages("PerformanceAnalytics") # youll need this
library("PerformanceAnalytics") # load it

##  and now for the magic
chart.Correlation(NIHS.data[c(4:10], bg=NIHS.data$SEX, pch=21)

## Great huh?
## what it does: scatterplot as above, but gives significance, correlation stat,
## histograms, trend lines, and group colouring (bg=, is 'by group')
## pch= is the point character
```

It's actually quite amazing just how much information can be crammed into one plot using the above function! As you can see, this is an incredibly useful tool for initial data exploration and correlation analysis.

Further exploration: 3d scatter plots

For those so inclined I've also included a few examples of 3d scatterplots in R. Feel free to have a play around with the code and data below.

```
## 3d scatter plot
install.packages("rgl")
require("rgl")
# plot3d(x, y, z, type="")
plot3d(BMI, LDL_C, SBP, type="p")

## another package for 3d scatter plotting:
install.packages("scatterplot3d")
require("scatterplot3d")
# scatterplot3d(x, y, z, type="")
Scatterplot3d(BMI, LDL_C, SBP, pch=16, highlight.3d=TRUE, type="h", main="3d Scatterplot")
```

Cross-tabs, chi square and t tests

There are a multitude of various statistical tests that can be run in R (the start of this section details a mere few of these). For this last part of the BMI analysis we're going to run three tests in particular: cross-tabs, chi square, and t tests.

Firstly we'll start by creating a new discrete variable for BMI. The cut-off here will be the mean plus one, so those below are group1 (low BMI) and those above are group2 (high BMI). This is the first introduction to an `ifelse` statement in R[27], follow along below and I'll explain how it's working.

```
# create a new discrete variable for BMI (those over/under mean+1)
NIHS.data$BMIcat = ifelse(NIHS.data$BMI >=27.92, 2, 1)
```

Right, pretty straight forward eh? All we're doing is defining a new variable (`BMIcat`) and filling it with data based on a set of rules. Those rules state that if BMI for an individual is equal to or above 27.92 then score them as 2 (group2), if they are not then score them as 1 (group1).

We can check that this variable has been created:

```
# check new variable BMIcat
head(NIHS.data)
NIHS.data$BMIcat
```

Yep, all looks good (hopefully!). You should see the new variable at the end of the `NIHS.data` object, and it should be populated with 1's and 2's.

Now that we have our discrete BMI variable we can create a table comparing BMI to Exercise (the variable PHYS_ACT represents those that exercise regularly [1] and those that don't [0]).

```
mytable <- table(BMIcat, PHYS_ACT)
mytable  # displays the cross tab/contingency table
```

Straight away this gives us an overview of the frequency of each category.

We can visualise this table using a barplot:

```
barplot(mytable, beside=T, ylim=c(0,300), space=c(0,0.5), border=TRUE, legend=c("BMI.low",
"BMI.high"), ylab="Frequency")
```

NOTE: legend provides labels for the category bars.

[27] Don't freak out, they're actually very useful.

Here we will look at several functions for running a chi squared test for independence. The first is the inbuilt function `chisq.test()`:

```
# example chi square tests
chisq.test(mytable)
```

So in this case we're testing the null hypothesis that BMIcat is independent of the exercise level (PHYS_Act) in NIHS. *What conclusion did we come to?*

For this next iteration of the chi test we'll need to download and install the Hmisc package.

```
# the chiSquare function below will require the Hmisc package
Install.packages("Hmisc")
require("Hmisc")

chiSquare(BMIcat~PHYS_ACT)
chiSquare(BMIcat~DIET)
chiSquare(BMIcat~MetS)
chiSquare(MetS~DIET)
```

Hopefully you'll notice that this is identical to the inbuilt chi square function, with a bit of extra information.

The last test we will look at is the t test. This is very simple to implement and follows the formula: `t.test(X~Y, data=mydata)` – where X=BMI and Y= a categorical trait that we want to test the means of - (obviously data=NIHS.data in our example).

```
# examples of t.test function:
t.test(BMI~PHYS_ACT, data=NIHS.data)
t.test(BMI~DIET, data=NIHS.data)
t.test(BMI~MetS, data=NIHS.data)
t.test(BMI~SMOKE, data=NIHS.data)
t.test(BMI~SEX, data=NIHS.data)
```

To visualise the possible difference of means we can use the boxplot() function:

```
# use the boxplot to view difference of means
boxplot(BMI~PHYS_ACT)
boxplot(BMI~MetS)
boxplot(BMI~SMOKE)
```

Feel free to have a play and explore other variables using the `t.test()` function:

```
t.test(T2D_risk~PHYS_ACT, data=NIHS.data)
t.test(BF~MetS, data=NIHS.data)
t.test(BF~HYPTEN, data=NIHS.data)
```

That concludes this brief introduction to basic stats in R for biomedical researchers. Continue to play with the different functions and search CRAN and other websites for potential packages that offer particular statistical tests and models that you might require. When you're ready move on to exercise 3 to further explore and enhance your understanding of stats in R.

Exercise 3: Explore and analyse data from the NIHS/NIES

For the last two exercise's I'm going to let you have free reign – I'm not providing any code at this point. I want to see what you've taken in, and how you're beginning to understand the data and its structure.

So as the title for Exercise 3 suggests, you should further explore the NI datasets. So reload back into R the files that we were using (the ones created in Exercise 2 and 3). Then it's up to you to explore the possible statistical tests and analyses to apply. There is no right and wrong answer, just have fun. If you get stuck, or come across any errors, shout out.

Here are a few hints/suggestions to get you started (also remember what we did in the previous example):

```
## exercise 3
# basic stats summary
summary(NIHS2009.pheno)

# correlations
cor(NIHS2009.pheno$Adj_SBP, NIHS2009.pheno$Adj_DBP, use="na.or.complete")
cor(NIHS2009.phenoWEIGHT, NIHS2009.phenoBMI, use="na.or.complete")

# t-tests
t.test(NIHS2000.pheno$SBP_2000, Adj_SBP)
```

Section 4: Basic Plotting/Graphing in R

This last section of the workshop introduces the concepts required to start you on your way to producing plots in R. R is renowned for its ability to output publication quality graphs, check out this repository if you'd like an example of the wide array of possible graphs:
http://addictedtor.free.fr/graphiques/

As you will notice in the above gallery (and the multitude of others available), the range of plots go from simple to ridiculously complex – or one line of code to several pages.[28] As you'll see most of the time starting simple - using `plot(mydata)` - will give you an initial starting point. Then it's only a matter of 'beautifying' your graph using the lower-level functions and `par()`.

In this section we are going to run through a variety of example plots, and then build upon them using `par()` and low-level functions. We'll cover how to output plots to vary file types, including .pdf, .png, .jpeg, etc. This is demonstrated using code in R, but I will also show you how RStudio handle the output of your graphs. I've also included the R-colour chart[29] at the end of this section. This chart is extremely useful when you want to use an array of different colours in your plot (or just want to be different).

Let's get started.

Plotting in R

There are three basic groups of plotting functions in R:

- high-level plots
- low-level plots
- the layout command `par()`

Basically, a high-level plot function creates a complete plot and a low-level plot function adds to an existing plot (one created using high-end functions). The layout command `par()` essentially sets up the 'style' of the plot(s) – you may have multiple plots in the same space. The `par()` function allows you to adjust almost everything, i.e. size of the text, colours, titles, scales and axis etc.

It's time to plot your first graph. Here we will plot car weight vs miles per gallon using the `plot()` command to generate a scatter plot.

Example: scatter plot using mtcars data

```
# first graph in R
attach(mtcars)          # attaches mtcars dataset
plot(wt,mpg)            # plots weight vs miles per gallon
```

Nice! You should see the graph appear in the 'plot' window of RStudio. You can enlarge this by pressing the 'zoom' option. It's a good time to note that in base R your plots are overwritten – each time you create a plot it replaces the previous one. In RStudio this doesn't happen, you may move between them using the forward and back arrows in the 'plot' tab.

[28] Hopefully you're now starting to think in terms of amount of code required, if so, welcome to the dark side.
[29] If I haven't made a note of it before, sometimes you'll need to be careful about the use of color and colour. Most package designers take into account that not everybody is American and allow use of both spellings, but sometimes you'll have to use one or the other. This is easy to check with R help.

Right, that was a good start but it's looking a bit plain at the moment. Let's add a few features to the graph by using some of the lower-level functions.

Example: scatter plot using mtcars data with added features

```
# 'fleshing' out your graph
plot(wt, mpg, xlab="miles per gallon", ylab="weight", pch=19)  # plots the data
abline(lm(mpg~wt))                    # adds line from regression model
title("Regression of MPG on Weight")  # adds a title
detach(mtcars)                        # detaches mtcars dataset
```

That's much better. You can see we added several parameters to further annotate the graph (`xlab`, `ylab`, and `pch`). You can find a detailed list of the lower-level parameters in R help, and I have also included a few commonly used functions in the cheat sheet at the end of this manual.

Here are a few more examples from Quick-R. While running through them see if you can figure out what lower-level functions are being implemented.

Example: boxplot using mtcars data

```
## boxplot of mpg by car cylinders
boxplot(mpg~cyl, data=mtcars, main="Car Milage Data",
        xlab="Number of Cylinders", ylab="Miles per Gallon")

## notched boxplot of Tooth Growth against 2 crossed factors
boxplot(len~supp*dose, data=ToothGrowth, notch=TRUE, main="Tooth Growth",
        xlab="Supplement and Dose", col=(c("gold", "darkgreen")))
```

Lower-level functions used: `main`, `xlab`, `ylab`, `col`, `notch`

In the following example we introduce a few more lower-level functions, namely the `legend` parameter.

Example: bar plot using mtcars data

```
# Simple Bar Plot
counts <- table(mtcars$gear)
barplot(counts, main="Car Distribution",
        xlab="Number of Gears")

# Simple Horizontal Bar Plot with Added Labels
counts <- table(mtcars$gear)
barplot(counts, main="Car Distribution", horiz=TRUE,
        names.arg=c("3 Gears", "4 Gears", "5 Gears"))

# Stacked Bar Plot with Colors and Legend
counts <- table(mtcars$vs, mtcars$gear)
barplot(counts, main="Car Distribution by Gears and VS",
        xlab="Number of Gears", col=c("darkblue","red"),
        legend = rownames(counts))

# Grouped Bar Plot
counts <- table(mtcars$vs, mtcars$gear)
barplot(counts, main="Car Distribution by Gears and VS",
        xlab="Number of Gears", col=c("darkblue","red"),
        legend = rownames(counts), beside=TRUE)

# Grouped Bar Plot (change label font size)
barplot(counts, main="Car Distribution by Gears and VS",
        xlab="Number of Gears", col=c("darkblue","red"),
        legend = rownames(counts), beside=TRUE, cex.names=2)
```

Lower-level functions used: `main`, `xlab`, `ylab`, `col`, `horiz`, `names.arg`, `legend`, `beside`, `cex.names`

Example: scatter plot using mtcars data

```
# Simple Scatterplot
attach(mtcars)
plot(wt, mpg, main="Scatterplot Example", xlab="Car Weight", ylab="Miles Per Gallon", pch=19)
# Add fit lines
abline(lm(mpg~wt), col="red")         # regression line (y~x)
lines(lowess(wt,mpg), col="blue")     # lowess line (x,y)

# Basic Scatterplot Matrix
pairs(~mpg+disp+drat+wt, data=mtcars, main="Simple Scatterplot Matrix")
```

Lower-level functions used: `main`, `xlab`, `ylab`, `pch`, `abline`, `lines`

There are many more advanced plotting functions and a multitude of packages that specialise in producing high quality graphs. Hopefully this has been a small taste of what R has to offer in terms of high and low level plotting option. Let's now explore the `par()` command.

The Power of par()

`par()` can be used to set or query graphical parameters. Parameters can be set by specifying them as arguments to par in `tag = value` form, or by passing them as a list of tagged values. For a list of parameters that may be used within par look at `?par` or `help(par)` – **Warning:** there are LOTS!

The benefits of using `par()` include:

- Having multiple graphs on the same 'page' – can set up a grid and align to this.
- Universal setting of lower-level parameters, i.e. axis scale, font size and colour
- Can quickly create complex looking plots

Let's start with a basic example:

```
# fitting labels
par(las=2)              # make label text perpendicular to axis
par(mar=c(5,8,4,2))     # increase y-axis margin

counts <- table(mtcars$gear)
barplot(counts, main="Car Distribution",horiz=TRUE,names.arg=c("3 Gears","4 Gears","5 Gears"),
        cex.names=0.8)
```

Here the par parameters `las` and `mar` are used.

Below is an example of plotting multiple graphs per page.[30]

```
par(mfrow=c(3,1))    # sets device so that it plots the 3 graphs on same 'page'

# Simple Bar Plot
counts <- table(mtcars$gear)
barplot(counts, main="Car Distribution", xlab="Number of Gears")

# Simple Horizontal Bar Plot with Added Labels
counts <- table(mtcars$gear)
barplot(counts, main="Car Distribution", horiz=TRUE,
        names.arg=c("3 Gears", "4 Gears", "5 Gears"))

# Stacked Bar Plot with Colors and Legend
counts <- table(mtcars$vs, mtcars$gear)
barplot(counts, main="Car Distribution by Gears and VS",
        xlab="Number of Gears", col=c("darkblue","red"),
        legend = rownames(counts))
```

[30] It's not the best example, but it illustrates the use of `par` option `mfrow`.

A more accurate example of the use of `mfrow` (**note:** this won't be reproducible on your system):

```
##############################################################################################
## Plotting the whole lot
##############################################################################################
## set pdf and dimensions
pdf("X_analysis_GenABELtesting_110908.pdf", width=17, height=9)
par(mfrow=c(3,1))    # sets so that it plots the 3 graphs on same page
## Plotting results RUN 1
plot(position, pval, col="blue", main="X analysis: Run 1 - no adjustment", xlab="bp position",
        ylab="-Log(p value", cex=1.2, sub="(X chr)", ylim=c(0,5))
axis(1, at=c(25, 50, 75, 100, 125, 150))
abline(h=1.5, col="red", lty=2)
abline(h=4, col="red", lty=2)
abline(v=64.1 , col="black", lty=2)
abline(v=67.7 , col="black", lty=2)
abline(v=137, col="black", lty=2)
abline(v=147, col="black", lty=2)
## Plotting results RUN 2
plot(position.mms, pval.mms, col="cadetblue", main="X analysis: Run 2 - mmscore",
        xlab="bp position", ylab="-Log(p value", cex=1.2, sub="(X chr)", ylim=c(0,5))
axis(1, at=c(25, 50, 75, 100, 125, 150))
abline(h=1.5, col="red", lty=2)
abline(h=4, col="red", lty=2)
abline(v=64.1 , col="black", lty=2)
abline(v=67.7 , col="black", lty=2)
abline(v=137, col="black", lty=2)
abline(v=147, col="black", lty=2)
## Plotting results RUN 3
plot(position.grs, pval.grs, col="darkolivegreen4", main="X analysis: Run 3 - GRAMMAS",
        xlab="bp position", ylab="-Log(p value", cex=1.2, sub="(X chr)", ylim=c(0,5))
axis(1, at=c(25, 50, 75, 100, 125, 150))
abline(h=1.5, col="red", lty=2)
abline(h=4, col="red", lty=2)
abline(v=64.1 , col="black", lty=2)
abline(v=67.7 , col="black", lty=2)
abline(v=137, col="black", lty=2)
abline(v=147, col="black", lty=2)
##
dev.off()
##############################################################################################
```

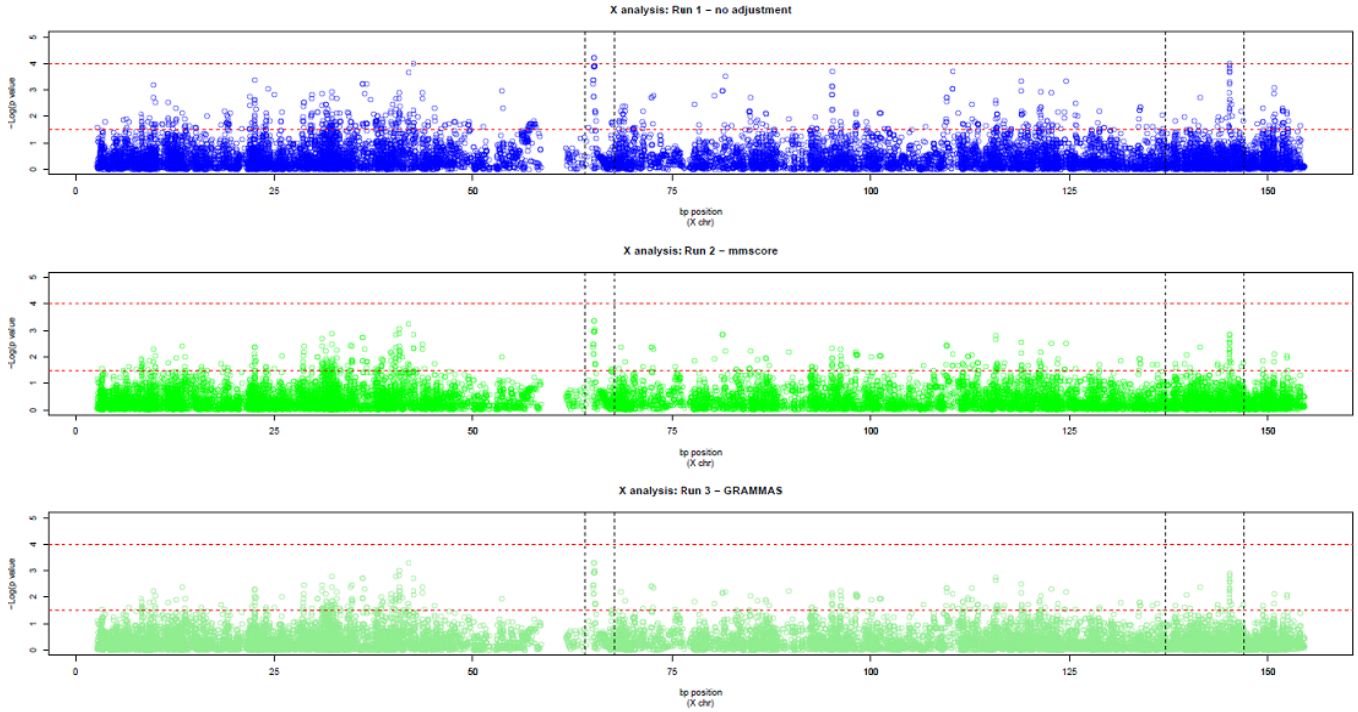

Figure 1: Example output using `par(mfrow=c(1,3))`.

Outputting Plots

Once you have generated you're wonderful plots you'll want to pass them round to everyone, here's several ways to achieve that.

Using RStudio

The first and easiest method is to use the 'export' option in the plot tab of RStudio. Here you can choose to export to .pdf, image (multiple types) and to the clipboard. When you select export as pdf or image you are presented with several options, such as size, quality, etc. You'll also get asked where you would like to save the file. Easy.

Sending plot output to file

Another option is to output your plot straight from the console. This requires opening the required graphics device[31], defining the type of file and size, running your plot code as usual, and then closing the device with `dev.off()`. [32] This might sound a bit weird, but follow the two examples below and it should become clearer.

Note: unless otherwise defined the file will be written in the current working directory.

Example: output plots from console

```
## this example sends output of the above graph to a pdf document
pdf("mtcars_plot.pdf")                # opens pdf plotter, and names the file
attach(mtcars)                        # attaches mtcars dataset
plot(wt, mpg)                         # plots the data
abline(lm(mpg~wt))                    # adds line from regression model
title("Regression of MPG on Weight")  # adds a title
detach(mtcars)                        # detaches mtcars dataset
dev.off()                             # turns off the plotting device

## we can do the same thing except send output to a png file
png("mtcars_plot.pdf")                # opens pdf plotter, and names the file
attach(mtcars)                        # attaches mtcars dataset
plot(wt, mpg)                         # plots the data
abline(lm(mpg~wt))                    # adds line from regression model
title("Regression of MPG on Weight")  # adds a title
detach(mtcars)                        # detaches mtcars dataset
dev.off()                             # turns off the plotting device
```

These are just two examples of possible output file types. Others include .jpeg, windows metafile, bitmap, etc. In all cases output will be saved in the current working directory unless directed elsewhere.

R colour chart

There are hundreds (657) of predefined colours available to you in R. These are used when plotting and defining line/point/text colours in your code. I've list several options for you to locate them.

To see the list of colours:

```
colours()
```

There is a very nice pdf with all the R colour information you could ever need located here: http://research.stowers-institute.org/efg/R/Color/Chart/ColorChart.pdf. This pdf lists all the colours along with their numbers and names.

[31] I haven't covered the inner workings of devices within R, it is really not vital to your productivity. However if you're interested the R help and manual go into lots of detail on this and many other subjects.

[32] Once familiar with using `dev` you'll be able to batch analyses if you require it, i.e. it becomes a possibility to write a script using `dev` to handle tables/graphs output whilst you're doing other things.

Exercise 4: Plotting results from the previous exercises

Right, the final Exercise of the day. Exercise 4 follows the same outline as the previous exercise; it's all up to you. So again using the NI data see if you can come up with some interesting and beautiful graphs. Feel free to explore as much as you want, you can even download a few external packages such as `lattice` and `ggplots` and have a play with them.

Again, remember that nothing is right or wrong, the best way to learn is by 'playing'/trial and error. If you need places to start look over the code from Section 4 and go from there. Also remember that there is lots of help available online.[33] I look forward to seeing what you come up with.

Here's a hint for some interesting statistical analysis combined with plotting (lots of information is displayed using this set of functions).

```
## setting up for some flash output
install.packages("PerformanceAnalytics")      # you'll need this
library("PerformanceAnalytics")               # load it

################this code is just to show what base R functions do
## check correlations
cor(iris[-5])        # investigate the correlation between variable
pairs(iris[-5])      # check scatter plot
# note: [-5] excludes the 5th column/variable, as it is not required
################

## … and now for the magic
chart.Correlation(iris[-5], bg=iris$Species, pch=21)
## Great huh?
## what it does: scatterplot as above, but give significance, correlation stat,
## histograms, trend lines, and group colouring (bg=, is 'by group')
## pch= is the point character
```

See if you can use this on the NI data.

Enjoy.

[33] There are also the people around you. ☺

Advanced Issues

Here I want to mention several issues that I have come across when working with R/RStudio on 'work' | institute computer systems (read: systems that are 'locked' down and you have very little freedom). There are ways around these issues to a certain extent, but the best advice I can give you is to befriend someone in IT and try to get it sorted at that end – this will save you a lot of time and grief. If this is not an option, then feel free to try some of these tips and tricks.

Note: *these tips/tricks will only be required if you are using R on a system that is not your own, or if the network you are using requires proxy, or admin privileges, i.e. I wouldn't recommend using the internet/proxy adjustment lines on your home desktop or laptop in less you know what you are doing!*

Proxies and package updates

Depending on the network options setup on the system you're working on, your mileage may vary when trying to download and install packages. If you are having issues here are several functions that I have found that might help out somewhat:

```
# Set or disable the use of Internet Explorer for Internet access.
# this can sometimes allow you to enter your site proxy details (read on...)
utils::setInternet2()

# set environmental condition for proxy access
# for most places the default port is 8080
Sys.setenv(http_proxy="http://myproxy.info.co.nz:8080 http_proxy_user=ask")
# you'll need proxy address and port number (and of course user name and password!)
```

Note: if you have a different port, get hold of IT and ask them which one you should be using.

Permissions and write protection

The default R library directory is within the main R installation (as mentioned previously). Sometimes, depending on the rights/privileges, this folder will be inaccessible and/or write protected – meaning that you won't be able to install packages there. Great. Well you might be thinking 'this isn't too much of an issue as I'm using RStudio', you'd be semi correct. With RStudio you can create a default library in the user directory of the system you are working on, but sometimes it might not be listed as the default library for package installation. Luckily we can set the default library with ease:

```
.libPaths()                                          # lists current library paths
.libPaths("C:/users/mbenton/R/win-library/2.13")     # will add desired location to .libPaths

# if all else fails this usually works:
install.packages(kinship2, lib="C:/users/mbenton/R/win-library/2.13")
# set lib="" to your desired library
```

Afterword

Phew, that well-and-truly brings us to the end of this workshop. I hope it has been useful and fairly straight forward to follow. As I keep mentioning we're only just scratching the surface of what you can do with R, but hopefully it opens your eyes to go out and explore further. The hardest part is getting over the initial hurdle; the fact that R is a programming language and there aren't any 'nice' buttons to click. But once you start to get a feel for how things work it's like the lights go on; error messages are no longer scary, you gain a deeper understanding of your data, and you'll start writing R scripts for numerous tasks.[34]

I'm hoping that it becomes possible in the future to offer a course or two that will delve deeper into Bioconductor and the packages/applications that are involved with genomics and bioinformatics analysis. I know there are a few people out there that are/have/will be doing this sort of analysis, and in my opinion this is where R really begins to shine.[35]

One of the things I've wanted to do for a while is create a communal blog or electronic workbook that everyone could access. This could be used as a place for people to post questions, problems, advice, useful links, resources, code, etc. For example, you're working on a piece of code and something is just not working, post on the blog and get the help of others. I know that there are many much bigger blogs out there, but it's nice to field questions to people you know initially.[36] I think now that we've been through one of these workshops as a group everyone will hopefully feel comfortable with each other – and it's nice with everyone at a very similar level of skill. This is just a thought; let me know what you all think. **UPDATE:** I followed through with the above idea and created a free blog at http://guru.forumotion.co.nz/. Please feel free to sign up and browse the currently available content, post questions, add helpful insights etc. I'm going to try and add a whole lot more content in the next month or so (as time allows…).

Well, I hope you enjoyed the workshop, learnt a little something and don't have too much of a sore head. All going well we hope to run more workshops in the near future.

Miles

P.S. any feedback is appreciated, such as; what was good, what was bad, too easy, too hard, couldn't understand, got sick of footnotes… ☺

Contact: miles.benton84@gmail.com

[34] Who knows, this might even lead you to other languages – and Unix/Linux based systems if you haven't used them before.

[35] **UPDATE (01-10-2012):** we are currently in planning stages for a series of genomics based workshops. These will be structured along the lines of: introduction to R and genomic analysis, manipulating genomic data, R: a foundation for microarray analysis, GWAS analysis in R, Gene expression analysis in R, Methylation analysis in R, batch scripting genomic analyses.

[36] Sometimes it saves a little embarrassment – old hands on the mailing lists can get quite sarcastic. ☺

Cheat Sheet

There are many R cheat sheets out there that provide brief summaries of functions and basic R operation (several of the better ones are listed in the resources section of this document). I have provided some of the commands that I think are essential/useful/interesting below, these are listed by category. For further details concerning any of these functions, consult the R Language help system by entering `?<function_name>` at the R system prompt (example: `?read.table`)

Miscellaneous

`q()` quit
`<-` assign
`=` assign
`m1[,2]` column 2 of matrix m1
`m1[,2:5]` or `m1[,c(2,3,4,5)]` columns 2:5 of m1
`m1[2,]` row 2 of m1
`m1[2:5,]` or `m1[c(2,3,4,5),]` rows 2:5 of m1
`NA` missing data
`is.na` true if data is missing

Help

`help(x)` get help with command x
`help.start()` start html browser help
`help(package = x)` help with the x package
`example(command1)` examples of command1
`data()` list of available data sets (these are examples from loaded packages)

Packages and R Library

`install.packages("package1")` installs package1 from CRAN into R library
`.libpaths()` get library location
`library()` see all installed packages
`library(kinship)` loads package kinship
`search()` see packages currently loaded
`sessionInfo()` see session info (includes R version, loaded packages, etc.)

Input/Output

`getwd()` See the current working directory
`setwd()` Set the current working directory
`dir()` shows files in the current working directory
`read.table()` Read in many kinds of data
`read.csv()` Read in comma delimited data
`read.delim()` Read in tab delimited data
`write.table()` Write tabular data in many formats
`scan()` Read in really huge datasets (faster but more complicated function)
`download.file(url1)` Download file from inputted address
`url.show(url1)` or `read.table.url(url1)` Remote read in of data

Create and manipulate data objects

`c()` Make a vector of specified elements, or add to a vector
`seq()` Numeric sequences
`rep()` Vectors of repeated values
`matrix()` 2-dimensional arrays
`array()` Multidimensional arrays
`data.frame()` Looks like a matrix, but some columns can be numeric, others character
`list()` Collections of different data objects ("cell arrays" in Matlab)
`ls()` Print a list of the all objects currently in memory
`rm()` Remove (erase) objects
`rm(list=ls())` Removes **ALL** loaded objects. **USE WITH CARE!!**
`t()` switch rows and columns (transpose)
`as.factor`, `as.list`, `as.matrix`, `as.character`, `as.vector`, `as.numeric`, `as.dataframe`
Conversion from one type to the other (if possible, i.e. correct dims etc.)
`is.factor`, `is.list`, `is.matrix`, `is.character`, `is.vector`, `is.numeric`, `is.dataframe`
what it is

Indexing

`which()` Find elements of a vector that satisfy a condition
`X %in% Y` Reports true if X exists in Y
`is.element()` Given two vectors, find elements that occur in both
`length()` Get the length of a vector
`dim()` Get or assign the dimensions of a matrix, data frame, or array
`ncol()` The number of columns in a matrix
`nrow()` The number of rows in a matrix
`colnames()` Get or assign column names of a vector
`rownames()` Get or assign row names of a vector
`names()` Get or assign the column names of a data frame or object names of a list

Database operations

`merge()` Merge or join data frames
`unique()` List of each unique element in a vector (even if they repeat)
`subset()` Pull out certain records in a data frame
`aggregate()` Do operations (means, sums) on columns of data frame given grouping variables
`stack()` Take repeated elements of a vector and put them in different columns
`reshape()` Like stack() and unstuck() but more flexible
`sort()` Sort a vector
`table()` Count the number of times individual values occur

Take sums/means/variances of rows/columns/arrays

`rowSums()` Just what it sounds like for 2-D matrices
`colSums()` Ditto
`apply()` Apply any function to specified dimensions of an array

Statistics

`max()`, `min()`, `mean()`, `median()`, `sum()`, `var()` as named
`summary(data.frame)` prints stats of data.frame

`rank()`, `sort()` rank and sort
`ave(x1,y1)` averages of x1 grouped by factor y1
`by()` apply function to data frame by factor
`apply(x1, n1, function1)` apply function1 (e.g. mean) to x by rows (n1=1) or columns (n2=2)
`table()` make a table
`tabulate()` tabulate a vector

Basic Statistical Analysis

`aov()`, `anova()`, `lm()`, `glm()` linear and nonlinear models, and anova
`t.test()` t test
`prop.test()`, `binom.test()` sign test
`chisq.test(x1)` chi-square test on matrix x1
`fisher.test()` Fisher exact test
`cor(a)` show correlations
`cor.test(a,b)` test correlation
`friedman.test()` Friedman test

Probability

`runif()` Draw random numbers from a uniform distribution
`rnorm()` Draw random numbers from a normal distribution
`dnorm()` Normal density function (many other distributions available)
`sample()` Sample elements from a vector

Control structures

`for()` Loops
`break()` Escape a for loop
`if()`, `else()` Self-explanatory
`ifelse(x, y, z)` like above, if x is true then y else z

Text manipulation

`paste()` Concatenate strings
`grep()` Find regular expressions in a character vector
`substr()` Extract or replace pieces of a character vector

General plotting functions

`plot()` Scatter and line plots
`plot(x,y)` Bivariate plot of x (on x-axis) by y (on y-axis)
`matplot()` Literally “matrix plot”, good for time series of many responses
`pairs(matrix)` scatter plot
`pie()` pie-chart
`coplot()` conditional plot
`hist()` Histograms
`barplot()` Barplots
`qqplot()` quantile-quantile plot
`qqnorm()`, `qqline()` fit normal distribution
`image()` Good for “maps” of matrices

Some common plotting parameters

Note: parameters are used inside functions (i.e. `plot(x, y, main="Scatter plot:x vs y")`

`add=FALSE` if TRUE superposes the plot on the previous on (if it exists)
`axes=TRUE` if FALSE does not draw the axes and box
`type="p"` specifies type of plot
`xlim=`, `ylim=` specifies the lower and upper limits of the axes
`xlab=""`, `ylab=""` annotates the axes
`main=""` main title
`sub=""` sub-title (below main, in smaller font)

Low-level Plotting Commands

`axis()` Create a "custom" plot axis
`points(x,y)` adds points
`text()` Write text in a plot
`mtext()` Write text in the margins of a plot
`abline(a,b)` draws a line of slope b and intercept a
`abline(h=y)` draws a horizontal line at ordinate y
`abline(v=x)` draws a vertical line at ordinate x
`abline(lm.object)` draws the regression line given by lm.object (obtained with `lm()`)
`rect(x1, y1, x2, y2)` draws a rectangle with given coordinates
`legend(x, y, legend)` draws a legend at given location
`title()` adds a title

Plot outputs

`pdf()` Create the figure as a pdf file
`win.metafile()` Create the figure as a windows metafile
`png()` Create as a png file
`jpeg()` Create as a jpeg
`bmp()` Create as a bitmap file
`postscript()` Create as a postscript file
`dev.off()` Turns off open plotting device

Arguments within par(), the graphics parameters list

`adj` controls text justification
`bg` specifies the background colour
`col` controls the colour of symbols and lines
`font` controls the style of text (integer with R values)
`las` controls the orientation of the axis label
`lty` controls the type of lines (i.e. normal, dashed, dotted)
`lwd` controls the width of lines
`mfrow` Create a multipanel plot (defines by rows, i.e. have 3 plots – one on top the other)
`mfcol` Create a multipanel plot (defines by columns, i.e. could have 3 plots side by side)
`new` Add a new plot to an existing plot
`mar` Lines of space in margin of each plotting panel
`oma` Lines of space in the outer margin of the plot
`tck` controls length of tick marks
`tcl` Length of axis tick marks
`cex` Magnification of symbols and labels (also see cex.axis(), etc)
`mgp` Controls distance between the axis and axis labels and title