The graphicsQC package

Quality Control for Graphics in R

Stephen Gardiner

Supervised by Dr. Paul Murrell

Department Of Statistics
The University of Auckland
BSc(Hons) Project

November 2008

Abstract

The **graphicsQC** package is a new R package developed for extending Quality Control for Graphics in R. It is capable of evaluating arbitrary code to produce plots in different file formats, while recording information about them. Sets of these plots are then able to be compared, with plots of differences produced (when available). Lastly, information about these comparisons are produced in an HTML report.

Contents

Chapter 1

Introduction

There has been much work into the concept of Quality Control for software (Bourque and Dupuis, 2004). There are currently tools in R which ensure that R code will run without catastrophic failure, but fewer tools to check that the output from the code is correct, especially graphics output. Thus, the aim of this research has been to establish new methods of ensuring Quality Control for graphics in R. The **graphicsQC** package has been created for this, and this report aims to describe how the package works, some features of it, and the reasoning behind some of the design choices.

In Chapter 2, the process of how plots are created and the steps taken to record the plots and related information is described.

Chapter 3 describes how the comparison between sets of plots is performed and some extra features the package provides for this.

Chapter 4 explains how reports are generated based on the comparisons.

1.1 Measuring software quality: QC and Regression testing

Quality Control (QC), or *testing*, is used to ensure quality of the output of software (as opposed to Quality Assurance (QA) which is involved with ensuring the correctness of the *process* producing the output). It is focused on ensuring that the software does not produce any errors (*crash*), and produces the correct output. Specifically, QC is useful for the detection of problems (*bugs*). In terms of graphics, assessing the correctness of output cannot (initially) be determined without the use of an 'expert observer'. The process of testing whether output is correct or not will be referred to as *validation*.

Once initial, correct output, has been produced one can test future outputs against this initial output. This type of testing is known as *regression testing*. The initial output is referred to as *control output*. In the case where this control output is known to be correct, that is if the graphics output has been validated, then this control output will be referred to as *model output*. After this model output has been created, changes to the source code of the software can be made, and the output produced again. This second set of output is referred to as *test output*. By comparing this test output against the control output, any changes in the output can be identified. If there are no differences between the

two, the test output can be validated as being correct. If a change is expected, then an expert observer is required to assess whether only the changes which were expected occurred, and are correct. If this is the case, the test output now becomes the control output for future testing.

1.2 Existing QC functions in R

There are currently a variety of methods used for quality control in R (Hornik, 2002). When writing packages, there is a `R CMD check` command which will perform a multitude of quality control tests on a package (R Foundation for Statistical Computing, 2008). Among these tests are checks of the output produced by the code. Any example code contained in the documentation of the package is evaluated to ensure that it does not crash or produce any errors. If a "tests" directory exists within the package, it will evaluate the code in any `.R` files in the directory, making sure that they do not produce any errors or crash, and then compare the output to corresponding `.Rout.save` files if they exist. Lastly, it will evaluate code in any package vignettes if they exist, to test that the code will not crash or produce any errors. This testing can be manually performed on any package at any time, however this process is also commonly automated. Examples of this exist on both the CRAN (The Comprehensive R Archive Network) and R-Forge websites, which both offer nightly checks and builds of the packages they host.

At R's core, there are also some hard-coded regression tests which can be run from a 'source' distribution of R via the `make check` command. These mainly consist of testing core elements, such as arithmetic tests, random number generation tests, and so on. They generally consist of a `.R` file containing test code to be evaluated, with a corresponding `.Rout.save` file which contains model output. There are however some graphics regression tests which are also run. These involve running R code which will open a PostScript device and run code which will produce some plots, and then use the GNU `diff` utility to compare the plots to some supplied model `.ps.save` plots. This facility is hard-coded into `make check` and is not performed by `R CMD check`. Thus the current level of quality control for graphics in R is very limited and is available only for a few predefined plots, and only when using a source distribution of R.

1.3 An early attempt at QC for graphics in R

An early attempt at graphics testing for R is a package similarly named **graphicsQC** written by Murrell and Hornik (2003). It is essentially a proof-of-concept package, and is very limited in usability and functionality. It is limited to only running regression tests on the example code in given functions in R. It does not record or return any information pertaining to the plots (such as which directory the plots are being stored), and had very little error checking. For most of the supported file formats, if any differences were detected, a vector naming the test files with differences were returned, with no other information. For the files in the `pbm` format, an `xor` operation is performed on the plots and a difference plot produced which gives a visual representation of the difference between the plots, however this is not supported for any other file format.

Due to these limitations, the package required a complete re-write to become more useful. Some features necessitating this are the ability to plot any arbitrary expressions, which can then be extended to plotting files and also examples from functions. As this occurs, it would also be useful to appropriately record any warnings which occur, and then possibly difference these as they may be useful for discovering why plots differ. It is also necessary to be able to produce plots highlighting the differences between two plots for all of the available file formats. This is slightly in contrast with the previous implementation where the plots were only of the *differences* between the plots, whereas it would be more desirable to see the original plot, but with the differences highlighted. Another necessary feature is the ability to record information relating to the plots, such as which plots were created in which directory, which call created the plots, and so on. The current **graphicsQC** package is a complete re-write of the previous implementation.

1.4 Outline of graphicsQC

The nature of what's required of the package reflects well with its design. The first component of regression testing involves creating control output. This is accomplished by one of the three functions `plotExpr()`, `plotFile()` and `plotFunction()`. These evaluate and plot arbitrary expressions, code within files, and example code from functions respectively. They also record information about the plots they produce in 'logs', including any warnings or errors that may have occurred. These functions are used for creation of both the control and test output and are further described in Chapter 2.

Once the control and test output have been created, they need to be compared for differences. For this, there is the `compare()` function. The `compare()` function will compare two sets of plots, for example two sets produced by `plotFunction()`. It can compare using the R objects of the logs, or paths to the resulting logs, or a mixture of these. It uses the GNU `diff` utility to do the comparisons, which can be assumed on all of R's supported platforms (R Foundation for Statistical Computing, 2008). It compares the plots and any warnings or errors for differences and then records the result of the comparison. The `compare()` function is further described in Chapter 3.

Lastly, it may be desirable to report about the plots and the comparisons. The function `writeReport()` accomplishes this by generating a HyperText Markup Language (HTML) page of the results of any plotting or comparison result. It is further discussed in Chapter 4.

Chapter 2

Plotting Arbitrary Expressions

For graphical output to be produced, expressions, or code, must first be evaluated. The ability to plot and test arbitrary code is a necessary feature in testing graphical output, for example if a 'user'[1] wishes to test graphical output after adding a new feature. It is imperative to store information related to plots that are produced for aiding the ability to perform comparisons. When dealing with code which produces graphical output, it is generally impossible to produce every possible output. Within R there are large amounts of example code exhibiting how a function works and how to use it. This example code is used to produce a sample of plots that can be produced by a given function.

2.1 Plotting expressions

For the task of evaluating arbitrary code to produce graphical output, the function `plotExpr()` was created. The main concept behind `plotExpr()` is to evaluate plotting code under chosen graphics devices and record which plots were produced and other related information such as operating system, date, directory, R version and so on. It does this by initially error checking its arguments, and then making a call to a function within the namespace, `evalPlotCode()`.

The `evalPlotCode()` function is responsible for evaluating code after opening an appropriate graphics device. It uses the `tryCatch` mechanism in R to ensure that the function can continue evaluation if there is an error in the code being evaluated. With the use of calling handlers, it is also able to 'catch' warnings and store them. If an error is encountered, this is also recorded and evaluation of the current set of expressions is stopped. This is intentional because an error in the code is likely to be something serious which will affect future expressions in the current set and possibly plots. However, if a warning is encountered, the warning will be recorded and evaluation will continue. How this information is recorded is discussed in section 2.3.

The call to `evalPlotCode()` is made within `plotExpr()` using the `lapply()` function. There are two advantages for this. First, the computation is

[1]The term 'user' here is used very loosely as it is likely to only be developers of R who would make changes to the way graphical output is produced.

vectorised, which R has been optimised for. Secondly, it ensures that `evalPlotCode()` gets called once for each file format for the expression, so any warnings or errors are captured separately by filetype. This is because some warnings may only occur on a certain graphics device and this ensures that the devices are treated separately to each other. Using many file formats is a distinct advantage in that all graphics formats can be tested against themselves to help identify whether changes in graphics output are simply due to changes in the code being evaluated, or whether there is a problem or change with a specific graphics device.

A special case to consider when evaluating plotting code is any code which will not actually produce any plots. Due the large variety of functions in R, and the ability to create your own on the fly, it is unfeasible to determine whether code will produce plots without first evaluating it. So initially `evalPlotCode()` will open the appropriate graphics device, evaluate the code, and then close the device. If no plot is produced, this will leave a 'blank' image in the chosen file format. These 'blank' images are not always blank in the sense that some information is written to file. This information differs by file format and so the resulting sizes of files will be different. These plots are not of interest when creating plots as they do not represent a plot produced by R. To deal with this, `plotExpr()` calls `generateBlankImages()` which generates 'blank' images in a temporary directory for the (supported) file formats which produce 'blank' files of non-zero size. The plots that are produced are then compared to these model 'blank' images, and removed if they are the same size, that is, completely blank. However, warnings and errors are still recorded in these situations.

A similar problem faced is that R cannot reliably determine how many plots will be produced from a given set of expressions, and so it is difficult to establish which plots were produced. The plots are named with a prefix according to what is specified as the `prefix` argument, along with a numbered suffix to identify each plot. That is, plots are created and then detected via the chosen prefix. As a consequence, care must be taken when choosing a `prefix` for the plots which is unique within a directory so that the plots that are created can be distinguished from other files in the directory. The chosen directory to produce plots in is checked prior to evaluating the code for any currently existing files that might be created by the function. A `clear` argument to the function is available which first clears the directory of any files with a name the same as any that might be created.

Thus, the arguments to `plotExpr` are of the order, `expr`, which is a character vector (or an expression object) of the expressions to evaluate which may produce graphical output. Next is `filetype`, which is used to specify which file formats the expressions should be evaluated in. The argument `path` is the path to place the plots and log file in. The `prefix` argument specifies the prefix to use when naming the files, with the prefix being followed by a numbered suffix. If files already exist in the chosen path and include names similar to those which might be created, the `clear` argument specifies whether these files should be removed before evaluating any code or not. The resulting object is given the class `qcPlotExprResult`.

An example is given below.

```
> first <- plotExpr(expr = c("y <- 10", "x <- 1", "plot(x:y)"),
                    filetype = c("pdf", "png"),
```

```
                    path = "exampleDir",
                    prefix = "firstExample")
> first
plotExpr Result:
Call:    plotExpr(expr = c("y <- 10", "x <- 1", "plot(x:y)"),
filetype = c("pdf", "png"), path = "exampleDir", prefix =
"firstExample")
R version:    R version 2.6.2 (2008-02-08)
Directory:    /home/stephen/graphicsqc/notes/report/exampleDir
Filename:     firstExample-log.xml
Formats:
  pdf : Plots: firstExample-1.pdf
  png : Plots: firstExample-1.png
```

2.2 Plotting files and functions

A function for plotting arbitrary expressions has already been defined. It can be seen that plotting files or functions is simply an extension of this, where multiple files may each correspond to a different 'expression', and likewise for functions where each function corresponds to a different expression. This is particularly true for functions, where the plots for each function are generated by a single call to `example()` of each function.

The two functions that provide these facilities are `plotFile()` and `plotFunction()`. They both essentially consist of checking their arguments, then generating multiple calls to `plotExpr()` for each file or function they are called with respectively. Thus the resulting objects contain lists of `qcPlotExprResult` objects. These `plotFile()` and `plotFunction()` calls are actually important in themselves as they need to refer to a grouping of `qcPlotExprResult` objects. Thus they also contain their own set of information similar to those stored in `qcPlotExprResult` objects. These results are classed as `qcPlotFileResult` and `qcPlotFunResult` respectively.

Conceptually, this is easily extended to the ability to plot entire packages as `qcPlotPackageResult` would consist of multiple sets of `qcPlotFileResult` and `qcPlotFunResult` objects. The ability to plot entire packages, for example `plotPackage()`, is not yet entirely implemented, but a close approximation can be made as shown in Chapter 5.The resulting classing structure is shown in Figure 2.1

How these are recorded as logs is described in the next section.

An example of plotting the `barplot()` function is given below.

```
 > bplot <- plotFunction(barplot,
                         filetype = c("pdf", "ps", "png"),
                         path = "barplot")
```

2.3 Logs of plots created

When creating plots, it is very useful to store information about them. Primarily, information such as which plots were produced in which file formats, and

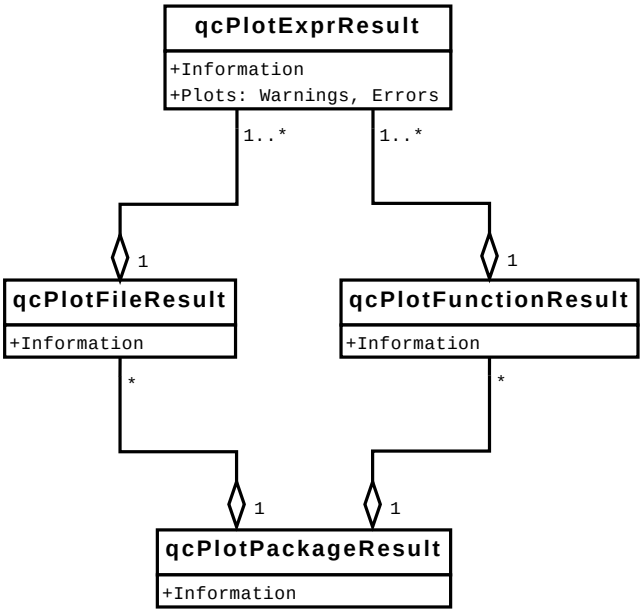


Figure 2.1: Classing structure for 'plot' objects

which warnings or errors were generated within these file formats is of interest. However information such as operating system, date, R version, and which call was used to create the plots is also useful. All of this information cannot be stored as a part of each plot, so a separate log file must be created.

As it is only text that needs to be stored, it makes sense to store this information in a text-based format, especially as it will be easier to read log files in the future. The logs were chosen to be stored using XML which is described in the next section. In order to work with XML documents in R, the **XML** package has been used (Temple Lang, 2001).

2.3.1 The XML Language

The e**X**tensible **M**arkup **L**anguage (XML) is a markup language for documents containing structured information. It is extensible due to the ability for users to define their own elements. It is an 'open standard' which agrees with the open nature of R. Some of the main advantages for using XML are its extensibility for user-defined elements, it is self-documenting, platform independent, relatively human-legible, and internationally recognised. This allows for great flexibility to store structured logs, as well as ease of reading in stored logs. A disadvantage of XML is that it is very verbose, and so can take up more disk space than if the log was stored using a binary format. This effect is considered to be negligible however, since the disk space required for plots will far outweigh the disk space required to store lists of their names in XML.

2.3.2 `plotExpr()` XML logs

The logs produced by `plotExpr()` consist of two parts. The first part consists of information related to the plots, such as operating system, R version, date, and call. The second part consists of the results of evaluating the code, such as the plots produced by the code, listed by file format, including warnings or errors.

The log created from section 2.1 (firstExample-log.xml) is given below.

```
<?xml version="1.0"?>
<qcPlotExprResult>
 <info>
  <OS>unix</OS>
  <Rver>R version 2.6.2 (2008-02-08)</Rver>
  <date>Thu Nov 13 21:43:29 2008</date>
  <call>
   <![CDATA[
plotExpr(expr = c("y <- 10", "x <- 1", "plot(x:y)"), filetype =
c("pdf",     "png"), path = "exampleDir", prefix =
"firstExample")   ]]>
  </call>
  <directory>/home/stephen/graphicsqc/notes/report/
exampleDir</directory>
  <logFilename>firstExample-log.xml</logFilename>
 </info>
 <plots type="pdf">
  <warnings/>
  <error/>
  <plot>firstExample-1.pdf</plot>
 </plots>
 <plots type="png">
  <warnings/>
  <error/>
  <plot>firstExample-1.png</plot>
 </plots>
</qcPlotExprResult>
```

The XML markup being used is fairly self-explanatory. Only 2 plots were created; one in PDF and one in PostScript, with neither having any warnings or errors. It is worth pointing out that the log includes its own `logFilename`. This is so the object in R knows where the file exists on disk.

Due to this completely open and platform independent way of storing this information, the process can be taken out of R at any point. For example if the user only wished to create the plots and then do something else with them, the information is stored in an easily accessible format for them.

2.3.3 `plotFile()` and `plotFunction()` XML logs

As `qcPlotFileResult` and `qcPlotFunResult` objects are simply lists of `qcPlotExprResult` objects with some extra information, the type of log files they create reflect this

structure. Both `qcPlotFileResult` and `qcPlotFunResult` files contain an information section, followed by the paths to all of the `qcPlotExprResult` files that they made. This is a more efficient usage of disk space as all of the information in the `qcPlotExprResult` files does not need to be repeated. It also has the advantage that no changes are needed to `plotExpr()` as it can continue producing the logs it already does, with `plotFile()` or `plotFunction()` only producing an extra log file to refer to all the produced logs. It is worth noting that the R representations of `qcPlotFileResult` and `qcPlotFunResult` objects contain complete information, that is, all of the information contained in the sub-`qcPlotExprResult` objects as well.

The log file produced from the `barplot()` example in section 2.2 is given below.

```
<?xml version="1.0"?>
<qcPlotFunResult>
 <info>
  <OS>unix</OS>
  <Rver>R version 2.6.2 (2008-02-08)</Rver>
  <date>Thu Nov 13 21:43:29 2008</date>
  <call>
   <![CDATA[
   plotFunction(barplot, filetype = c("pdf", "ps",
"png"), path = "barplot",     clear = TRUE)   ]]>
  </call>
  <directory>/home/stephen/graphicsqc/notes/report/barplot
</directory>
  <logFilename>barplot-funLog.xml</logFilename>
 </info>
 <qcPlotExprResult>/home/stephen/graphicsqc/notes/report/
barplot/barplot-log.xml</qcPlotExprResult>
</qcPlotFunResult>
```

Note that the `qcPlotExprResult` element consists only of the path to the corresponding log file that contains information about the plots produced.

The naming of these `qcPlotFileResult` and `qcPlotFunResult` files requires extra care when more than one file or function is specified. In the example given above, the filename was `barplot-funLog.xml`, as only one function was specified. If multiple files or functions are specified, the prefix used is chosen to be the first element in the `prefix` vector. Any other choice will inevitably result in a filename too long for the system.

Chapter 3

Comparing plots

In the case of quality control for graphics, it is not the code used to create the graphics that is of importance, but rather whether the final output has changed. Due to the nature of different file formats used for graphical output, comparisons can only be made within the same file formats. The major differences between some file formats are discussed in section 3.1.1. There is also an issue of pairing plots to be compared, for example if one function in a set of many includes a new plot in the test group, the functions should still be pairwise compared, but with the extra plot left out. How this is managed is described in section 3.1.2. Changes in warnings and errors can also provide clues as to why graphics output has changed, so these are compared along with the appropriate plots as described in section 3.1.3.

3.1 Comparing sets of expressions

Once two sets of plots have been produced, it is desirable to compare them for differences. The naïve approach is to display two plots side-by-side and visually compare them for differences, repeating this for the entire set. This is both ineffective and inefficient. It is very difficult for the human eye to discern small differences in images, especially when there are large amounts of detail in both images. It would also take a very long time if there were many plots.

A better solution involves using the GNU `diff` utility, which is assumed to be supported on all of R's supported platforms according to the R Coding Standards (R Foundation for Statistical Computing, 2008). GNU `diff` compares files to test if they are *exactly* the same. So once plots are appropriately paired, they can be 'diffed', to identify which plots are identical or different and narrow down which plots the user needs to examine. These differences may however still be extremely difficult for the user to detect, or take a long time until the user notices them. Because of this, ImageMagick software is also used. ImageMagick provides a `compare` utility that is used to produce a plot highlighting the differences between two plots. An example of this is given in section 5.2. This software is free (as in speech) so the source code is available for download, and is distributed as standard on some GNU/Linux distributions. It is however not necessary for `compare()` to work as primarily the differences are detected using `diff`.

The `compare()` function handles comparisons between all supported sets of expressions, for example between two `qcPlotFileResult` objects. The images that highlight differences are created by default if ImageMagick is installed and the plots are different. The `erase` argument to `compare()` specifies options for removing test output after performing the comparison. If `erase` is set to "files" or "all", that is, delete all the plots in the test group (leaving the log files), or delete all the files in the test group, then the plots highlighting the differences will not be produced. This is currently the only effect of the `erase` option as it does not yet remove any files.

Comparing the example from section 2.1 to itself should produce a result indicating the plots are identical.

```
> firstComparison <- compare(first, first)
> firstComparison
qcCompareExpr Result:
Call:
 compare(first, first)
                      Test                          Control             Results
R version: R version 2.6.2 (2008-02-08)  R version 2.6.2 (2008-02-08)
Directory: ...qc/notes/report/exampleDir ...qc/notes/report/exampleDir
Filename:  firstExample-log.xml          firstExample-log.xml
Format:
pdf        ...mpleDir/firstExample-1.pdf ...mpleDir/firstExample-1.pdf identical
png        ...mpleDir/firstExample-1.png ...mpleDir/firstExample-1.png identical
```

Note that this is not a particularly appealing way of presenting these results, especially if there were many files compared. A more effective report can be generated and is described in Chapter 4.

The results of comparing expressions are stored in XML logs in a similar structure to the `qcPlotExprResult` objects. Each `qcCompareExprResult` log contains its own information about what occurred, as well as information from the test and control groups. The log file created from the '`firstComparison`' example above is given below, with the information sections omitted for brevity's sake.

```
<compare type="pdf">
 <comparison controlFile="/home/stephen/graphicsqc/notes/
                          report/exampleDir/firstExample-1.pdf"
             testFile="/home/stephen/graphicsqc/notes/
                       report/exampleDir/firstExample-1.pdf">
  <result>identical</result>
  <diffFile></diffFile>
  <diffPlot></diffPlot>
 </comparison>
</compare>
<compare type="png">
 <comparison controlFile="/home/stephen/graphicsqc/notes/
                          report/exampleDir/firstExample-1.png"
             testFile="/home/stephen/graphicsqc/notes/
                       report/exampleDir/firstExample-1.png">
  <result>identical</result>
  <diffFile></diffFile>
  <diffPlot></diffPlot>
```

```
  </comparison>
 </compare>
 <unpaired>
  <test/>
  <control/>
 </unpaired>
```

As there was only one set of files to compare for each filetype, there is only one `comparison` element within each `compare` element. There were no unpaired files and no differences in warnings or errors — these are further discussed in section 3.1.2 and section 3.1.3 respectively.

The default placement of the log files and images highlighting the differences are in the `test` directory, however this can be changed by specifying a different `path` to the `compare()` function.

3.1.1 Bitmap vs. Text-based Formats

Bitmap formats are most commonly what the user sees, for example after generating a standard plot. When stored, they are relatively large files and limited to the resolution used at the time of saving. Murrell and Hornik (2003) also pointed out that the version of the third-party software used to create the image can produce slightly different output, and even differences in hardware setup (for example the platform being used). Currently the only supported Bitmap format is `png` (Portable Network Graphics).

Text-based formats tend to produce relatively small files at very high resolutions, and are generally more platform-independent. These however suffer a similar drawback to bitmap formats in that differences in the version of the third-party software used to create the image can result in different files produced. With these formats it is possible for these changes to have no visible result on the final image, for example a change in the internal structure of the file. The currently supported text-based formats are `ps` (PostScript) and `pdf` (Portable Document Format).

The GNU `diff` utility provides a textual output of the differences between files if they are different. For text-based formats, this information is useful and is possible for someone with knowledge of the format to interpret. Due to nature of bitmap formats, this is generally meaningless to humans. As such, when a difference between two files is detected, if the files are a text-based format, a `.diff` file is created giving the `diff` output of the differences, but this is not created for bitmap formats.

There is a special case when comparing two files in the `pdf` format. Files created in the `pdf` format contain some header information including the date and time the file was created. When creating two separate plots, the time each plot was created will inevitably be different from each other. As a result, when files are compared for differences, this header section is ignored, and the rest of the file is compared. If a difference is detected in the rest of the file, the differences in headers are included in the resulting `.diff` file. Due to this format dependent comparison, each format has its own function defined for how to perform the comparison. This is further discussed in section 3.4.

3.1.2 Pairing plots

When comparing plots produced by a set of expressions, it is important that each of the plots in the test group is appropriately paired with its corresponding plot in the control group. These pairings need to be done by file format as the number of plots may be different depending on the format, for example if an error prevented one format from producing some plots. It is also possible for all filetypes to have unpaired files if for example an extra plot was included in the test group and not in the control group. If the length of plots in a given filetype in one group is greater than the other, the last plots that do not match up are grouped into an 'unpaired' section that lists the plots that were unpaired by filetype.

It is also possible for entire file formats to be unpaired, for example if the control group consisted of plots only in the `png` format, whereas the test consisted of `png` and `pdf` plots. When this occurs, these plots are included in the 'unpaired' section as well as any warnings or errors that occurred because it is unknown whether these would have occurred under the same conditions as the control group. These are also reported on separately when the report is generated.

3.1.3 Comparing Warnings and Errors

When there are differences in the plots, differences in the warnings or errors often provide clues as to why this may be. It has already been discussed in section 2.1 that warnings and errors are recorded when code is evaluated. When the comparison between plots occurs, the warnings and errors for each filetype are also compared. If the warnings or errors are not identical to each other, all of the warnings or all of the errors (whichever had the difference) are reported for both the test and control groups for the filetype being compared. It is then up to the user to establish what the difference is. This is because ordering in the warnings or errors could also prove significant in why plots differ. If no differences between the warnings or errors are detected, then they are not reported in the comparison.

3.2 Comparing sets of files and functions

Comparing sets of files and functions work in a similar to way to plotting files and functions. The process is broken down into simpler, easier to manage chunks. This results in sets of `qcPlotExprResult` objects being compared. Each file or function corresponds to a single `qcPlotExprResult`, so these are just pairwise compared. As in the case of comparing expressions, there is an issue of unpaired files or functions. Currently, if one group contains more objects than the other, the group with less objects will 'recycle' the elements until the groups have the same number of objects to compare. A better option is to group these as 'unpaired', but this is not yet implemented.

When files and functions are compared, the comparison log file contains paths to the individual `qcCompareExprResult` files much the same as occurred for when files and functions were plotted. An example of a log file after comparing the '`bplot`' example to itself is given below.

```
> compareBplot <- compare(bplot, bplot)

<?xml version="1.0"?>
<qcCompareFunResult>
 <info>
  <OS>unix</OS>
  <Rver>R version 2.6.2 (2008-02-08)</Rver>
  <date>Thu Nov 13 21:43:30 2008</date>
  <call>
   <![CDATA[
   compare(bplot, bplot)   ]]>
  </call>
  <path>/home/stephen/graphicsqc/notes/report/barplot</path>
  <logFilename>barplot-compareFunLog.xml</logFilename>
  <testLog>/home/stephen/graphicsqc/notes/report/barplot/
           barplot-funLog.xml</testLog>
  <controlLog>/home/stephen/graphicsqc/notes/report/barplot/
              barplot-funLog.xml</controlLog>
 </info>
 <qcCompareExprResult>/home/stephen/graphicsqc/notes/report/
    barplot/barplot+barplot-compareExprLog.xml
 </qcCompareExprResult>
</qcCompareFunResult>
```

As there was only one function compared, there is only one `qcCompareExprResult` listed. As for when files and functions are plotted, the R representation of this contains complete information, so it will contain all of the information from the listed `qcCompareExprResult`.

3.3 Auto-detection of logs

One of the features the package provides when comparing (and even reporting) sets is the ability to auto-detect log files. An example of this is if two `qcPlotFunResult` objects needed to be compared, there are a couple of ways of specifying the groups. If the objects were created in the current R session, these can be given for the `test` and `control` arguments to `compare()`. If they were not, one can specify the path of the directory that contains the log file to compare to. The 'most important' log file will then be searched for in the directory and read as an R object and returned, where `qcPlotFunResult` and `qcPlotFileResult` objects are considered more important than `qcPlotExprResult` objects as they will contain them. Objects of class `qcPlotPackageResult` would be considered most important. The way for specifying these can be mixed-and-matched, for example the `test` group could be an R object while the `control` group could be specified as the path to the file to compare. This works as long as the resulting classes for `test` and `control` are the same. If they are not, an error is given. In the case where the log file is an ambiguous choice, an error is also given, for example if the same folder was given for the `test` group and for the `control` group, even if two separate files are detected, it is unclear which should be the test group and which should be the control. An example of this in use is given in section 5.2.

3.4 Extensibility for new file formats

In section 3.1.1 it was briefly discussed why it is necessary to have format dependent comparisons. As a result, each supported format has its own function defined for how to carry out the comparison. Due to how this has been coded, it is easily extensible for developers to add new formats. The function called to compare two plots is dependent on what the current format being compared is. For example if the current format was `png`, then the function that gets called to perform the comparison is `comparePNG()`. That is, the word `compare` with the format being compared in uppercase appended on, `PNG`. So if a new format were to be supported by **graphicsQC**, the only change necessary for comparing the format would be a new, appropriately named function.

Chapter 4

Generating Reports

Once comparisons have been performed, the results are stored in XML files. At this point, users of the package may wish to take the process entirely out of R or do anything they like with them. One option is to dynamically generate a report based on the comparisons. As the data are stored in XML files, there is a language defined that is useful for transforming XML documents into other types of documents. This language, XSL, and how it is used to transform the log files into HTML reports are described in the following sections.

4.1 HTML Reports

Depending on the comparison performed, there is generally a large amount of information to report on. The nature of how the log files are stored reflects well with the natural nature of HTML where different pages are connected to each other via links. For example `qcPlotFunResult` logs typically refer to other `qcCompareExprResult` logs. They also refer to different plots, which can be linked to and displayed within an HTML page. There is also a very natural way of transforming the log files from XML into HTML which is described in section 4.1.2. Web browsers are very widespread, so most users will be able to view HTML pages.

4.1.1 The XSL Language

The e**X**tensible **S**tylesheet **L**anguage (XSL) is a functional language used to define transformations of XML files into other formats. This is also sometimes referred to as XSLT. In XSL, styles, or templates, are created defining how to display elements. The XML document is traversed, with the appropriate templates applied to each element. It also makes use of the XPath language to address separate parts of the XML document, which is particularly useful when selecting nodes out of the order they are stored. The XSL language provides the efficiency and tools necessary to transform the XML logs into HTML reports.

4.1.2 Transforming XML logs into HTML reports

In order for the XML logs to be transformed into HTML reports, XSL stylesheets must be provided to define the transformation. Stylesheets for all of the objects

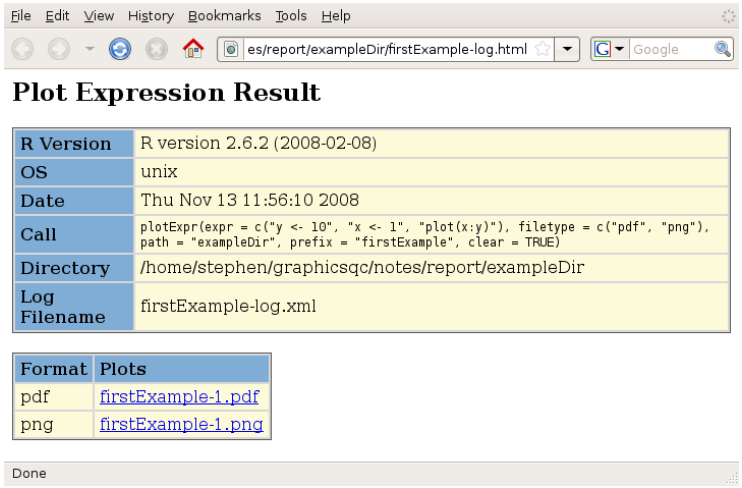


Figure 4.1: Report on a `qcPlotExprResult` object.

produced by the **graphicsQC** package are distributed with the package. These can be seen as being examples of a set of possible stylesheets, as the user of the package is able to specify their own stylesheets to use for the transformation.

In order to use XSLT from within R, the **Sxslt** package is required (Temple Lang, 2007). The package provides an interface to the libxslt translator, and also allows XSLT to use R functions. Currently, the **Sxslt** package is *not* supported on the Windows platform.

The function `writeReport()` generates an HTML report from its first argument, `qcResult`. If the function is given a 'more important' object, for example a `qcPlotFunResult` object, it will produce an HTML report for the current object, as well as all objects that the the current object refers to. This is because the page that is generated links to the sub-objects that are referred to in case the user wishes to see more detail. That is, for every XML log file that is referred to, a corresponding HTML file will be created with the same name, but with the `.html` extension. The `writeReport()` function is flexible in specifying which object to report on. It can accept either the R object, the path to the log file, or even auto-detect from a directory. If a directory is given, first any comparison log files are searched for, followed by log files created by plots, in decreasing order of 'importance'. Each object that is reported on has the appropriate stylesheet applied, however the default stylesheets can be overwritten by specifying the path to the desired stylesheets by the `xslStyleSheets` argument. An example of the report generated from the '`firstExample`' is given in figure 4.1. Note that there was only one plot produced for two file formats, with no warnings or errors. The table is sized appropriately for the number of plots and includes warnings or errors if there are any. An example of a report generated from a function comparison result is given in section 5.3.

```
> writeReport(first)
```

Chapter 5

A real example: grid

In revision 44417 of R an anisotropy correction was added to how xsplines are drawn. It was expected that this would only affect the bitmap formats (PNG), but not the text-based formats (PostScript and PDF). The magnitude of the effects were also not known. The **graphicsQC** package was used to compare revision 44417 of R against the previous, 44416, to establish what effects this change might have had on the **grid** package.

5.1 Plotting functions in the grid package

As a `plotPackage()` function has not been implemented, the **grid** package cannot be directly plotted. A close approximation is to simply plot all of the functions exported by the package. This would have to be done under both revisions of R as outputs from both will need to be compared. An example used under revision 44417 is given below.

```
> grid44417 <- plotFunction(ls("package:grid"),
                            filetype = c("pdf", "ps", "png"),
                            path="~/tests/R44417")
```

With a similar command used in revision 44416.

5.2 Comparing the grid functions

Once both the test and control groups have been created, they can be tested for differences. Revision 44416 is considered to be the control group, prior to the change, and 44417 to be the test group. The comparison will be performed under revision 44417, however the revision under which the comparison is performed will not make a difference. As the R object for the control group is in a different R session, the directory containing the control group can be given for the `control` argument and auto-detect will find the correct log. Thus, to do the comparison one could use a command such as that given below.

```
> gridCompare <- compare(test = grid44417,
                         control = "~/tests/R44416")
```

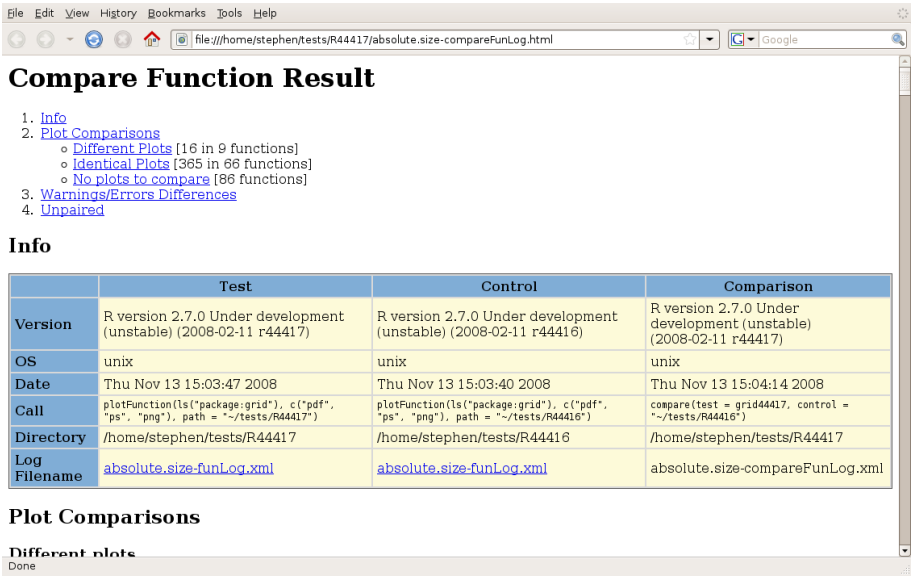


Figure 5.1: Report of **grid** comparison.

5.3 Reporting on the differences in grid

As the comparison is complete, `writeReport()` can be used on the resulting comparison object to generate an HTML report of the comparison. For example:

```
> writeReport(gridCompare)
[1] "/home/stephen/tests/R44417/absolute.size-compareFunLog.html"
```

which gives the path to the HTML report. Note that it has the prefix "absolute.size" because that is the first function in the list of functions which were compared.

Some of the output from the report can be seen in figures 5.1, 5.2, 5.3, 5.4 and 5.5. It is worth noting that all of the links are active, so the user can view reports from individual comparisons and the plots produced.

It was previously mentioned that some changes were expected. Some plots of functions did exhibit some change in the `png` format as shown in figure 5.2, but not in other formats, as shown in the break-down of the `arcCurvature()` comparison in figure 5.6. The `seekViewport()` function however had differences in all three formats being tested.

In figure 5.7, the test and control group plots for the `grid.xspline()` function are both shown side-by-side. They were reported as having a difference, but this might not be immediately obvious. The image highlighting the differences is given in figure 5.8. The image shows that all of the curves have been 'nudged' down slightly. It has been discovered this effect is connected with the users' setup for their `X` windowing system. When `X` is set up properly, there is no effect, but depending on on the `X` set up, the effect can be much more noticeable. As to why `seekViewport()` had changes in formats other than `png` is still being investigated.

Figure 5.2: Report of **grid** comparison.

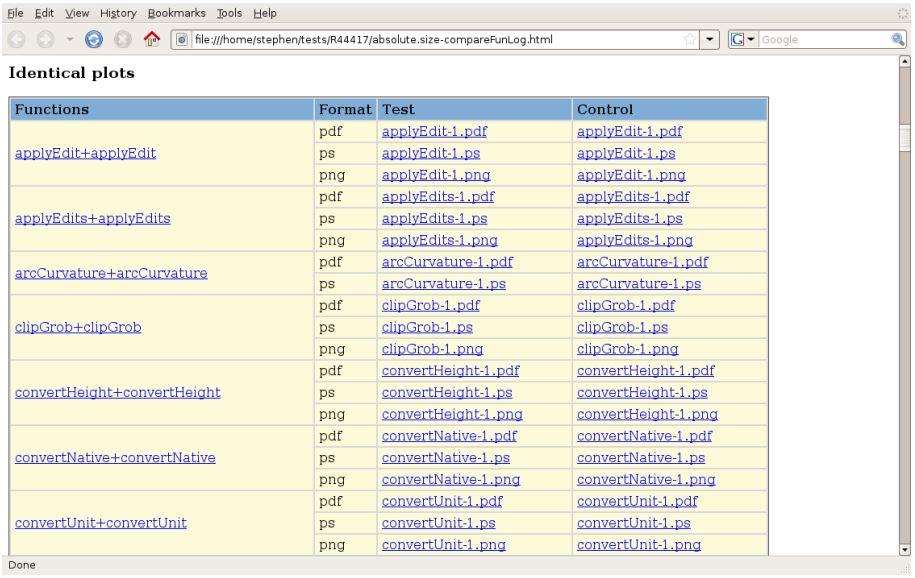


Figure 5.3: Report of **grid** comparison.

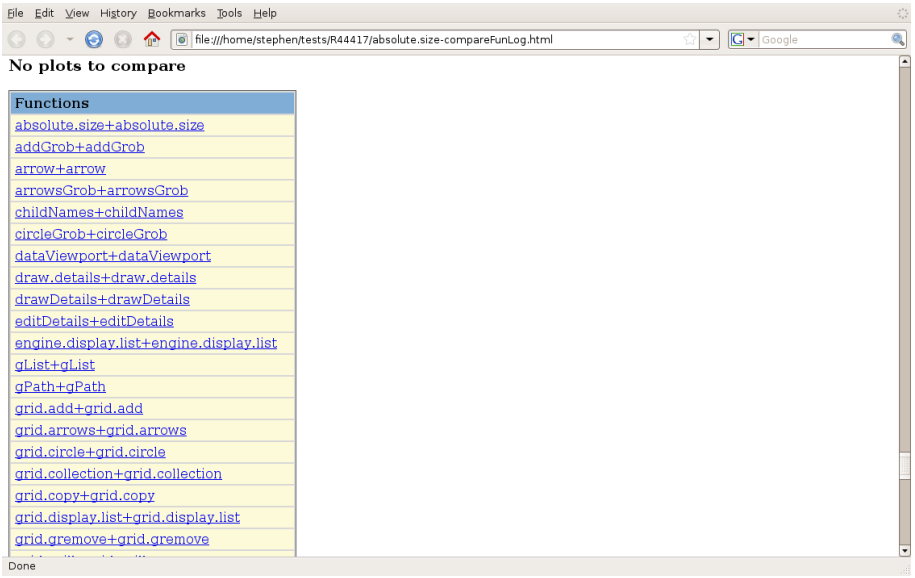


Figure 5.4: Report of **grid** comparison.

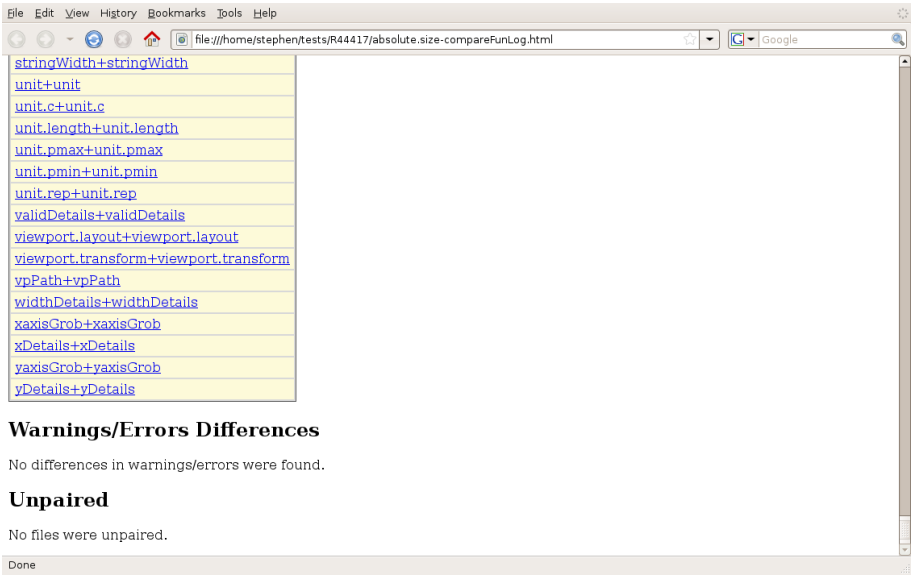
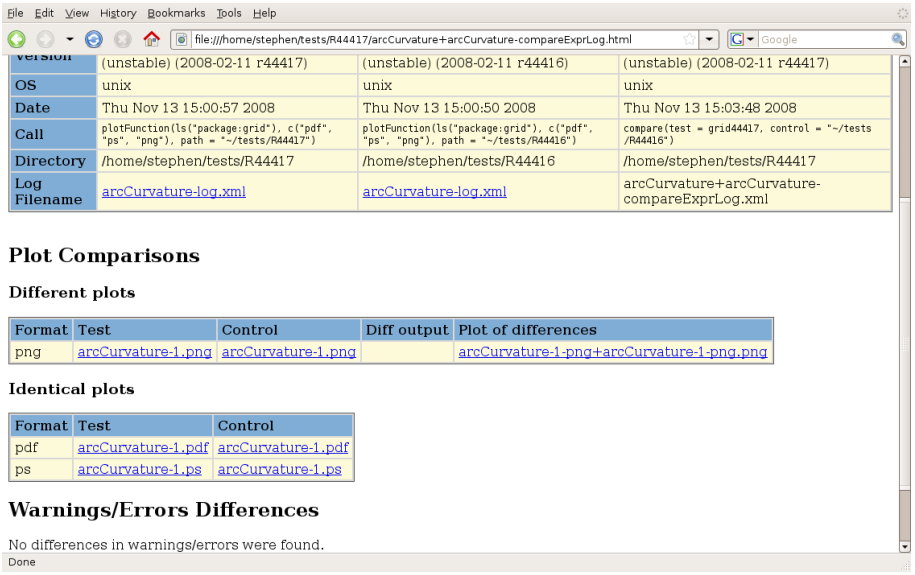


Figure 5.5: Report of **grid** comparison.



File Edit View History Bookmarks Tools Help

file:///home/stephen/tests/R44417/arcCurvature+arcCurvature-compareExprLog.html

Google

Version	(unstable) (2008-02-11 r44417)	(unstable) (2008-02-11 r44416)	(unstable) (2008-02-11 r44417)
OS	unix	unix	unix
Date	Thu Nov 13 15:00:57 2008	Thu Nov 13 15:00:50 2008	Thu Nov 13 15:03:48 2008
Call	plotFunction(ls("package:grid"), c("pdf", "ps", "png"), path = "~/tests/R44417")	plotFunction(ls("package:grid"), c("pdf", "ps", "png"), path = "~/tests/R44416")	compare(test = grid44417, control = "~/tests/R44416")
Directory	/home/stephen/tests/R44417	/home/stephen/tests/R44416	/home/stephen/tests/R44417
Log Filename	arcCurvature-log.xml	arcCurvature-log.xml	arcCurvature+arcCurvature-compareExprLog.xml

Plot Comparisons

Different plots

Format	Test	Control	Diff output	Plot of differences
png	arcCurvature-1.png	arcCurvature-1.png		arcCurvature-1-png+arcCurvature-1-png.png

Identical plots

Format	Test	Control
pdf	arcCurvature-1.pdf	arcCurvature-1.pdf
ps	arcCurvature-1.ps	arcCurvature-1.ps

Warnings/Errors Differences

No differences in warnings/errors were found.

Done

Figure 5.6: Report of `arcCurvature()` comparison.

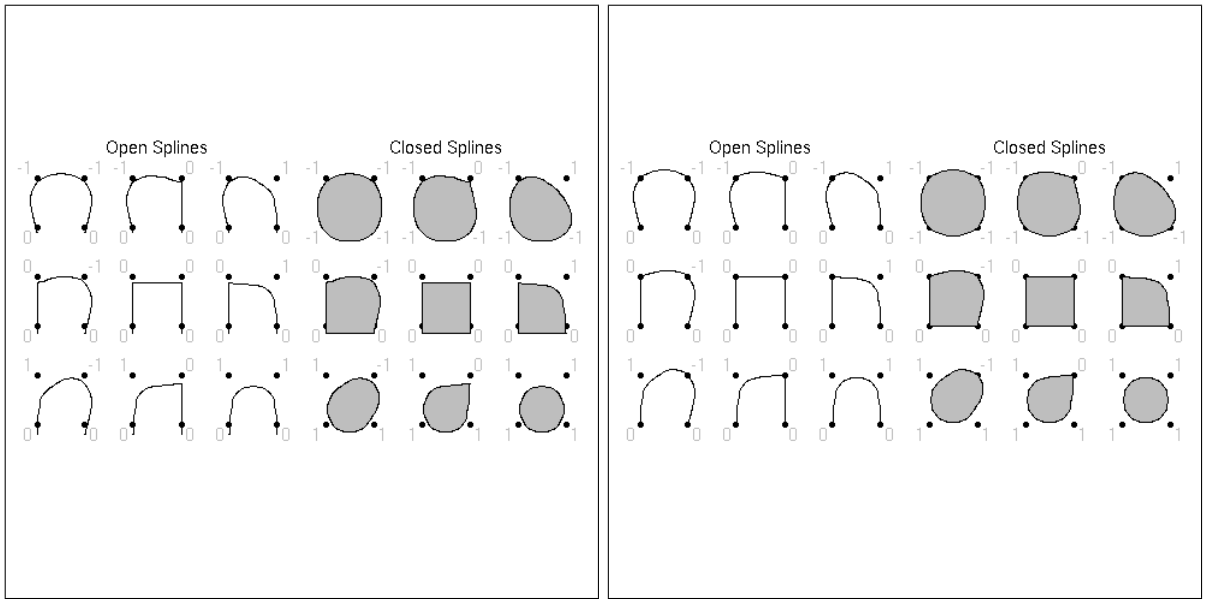


Figure 5.7: Respective test and control plots from `grid.xspline()`.

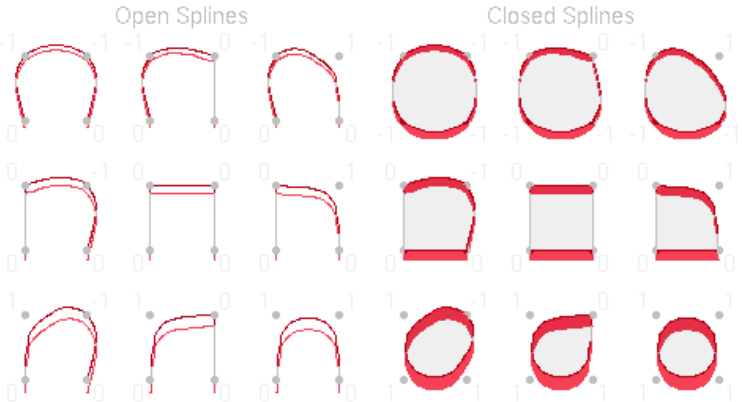


Figure 5.8: Differences in `grid.xspline()` between revisions.

Chapter 6

Summary and discussion

The **graphicsQC** package was developed to help ensure quality control for graphics in R. It is capable of evaluating arbitrary code to produce plots based on expressions, files, or examples from functions. It can then compare sets of plots for differences, and produce images highlighting the differences if there were any. HTML reports can be dynamically generated from the results of plots or comparisons. All the information related to plots and comparisons are stored in an open format, allowing for the ability to read the information in the future, and freedom of choice for users who may choose to report on the information in a different way.

An example was given showing how the package has already been useful for identifying changes in the graphics generated by the **grid** package. The example also showed that the package is only useful for detecting changes in graphics output, but cannot identify *why* changes occur.

There is still further work to be done in the **graphicsQC** package for improving its ability to ensure quality control for graphics in R. Mainly, the ability to plot and compare entire packages would greatly enhance the package's usability. Adding more graphics devices, and the ability to specify formats for each graphic device to use would add more cases for the package to check, so would be more useful in identifying changes. Both of these tasks have been made easier as the style of coding used throughout the package has been to allow for extensibility. Lastly, extending the package to work on platforms other than GNU/Linux to help ensure quality control for graphics in other platforms. This is considered a more long-term goal as it is hindered by the **XML** package which does not fully work on Mac OS, and by **Sxslt** which has not been ported to MS Windows.

Documentation

The latest version of the package can be installed from within R via the following command:

```
> install.packages("graphicsQC", repos="http://R-Forge.R-project.org")
```

Noting that the **XML** package is a dependency, so must also be installed, and **Sxslt** must also be installed if HTML reports are desired.

The latest version of the package can also be downloaded in unix directly from the Subversion repository, when issuing a command such as:

```
svn checkout svn://svn.r-forge.r-project.org/svnroot/graphicsqc
```

at the unix prompt.

To download the current revision at the time of printing (revision 60), the following command can be used:

```
svn checkout --revision 60 svn://svn.r-forge.r-project.org/svnroot/graphicsqc
```

The rest of the in-R documentation follows on the next page.

Bibliography

P. Bourque and R. Dupuis. Guide to the software engineering body of knowledge 2004 version. *Guide to the Software Engineering Body of Knowledge, 2004. SWEBOK*, Chapter 11, 2004.

Kurt Hornik. Tools and strategies for managing software library repositories. In *Statistics in an Era of Technological Change, Proceedings of the 2002 Joint Statistical Meetings*, pages 1490–1493, 2002.

Ross Ihaka and Robert Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996.

P. Murrell and K. Hornik. Quality assurance for graphics in R. In *Proceedings of the 3rd International Workshop on Distributed Statistical Computing*, pages 20–22, Vienna, Austria, March 2003. ISSN 1609-395X. Edited by Hornik K., Leisch, F. & Zeileis, A.

R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2008. URL `http://www.R-project.org`. ISBN 3-900051-07-0.

R: Writing R Extensions. R Foundation for Statistical Computing, Vienna, Austria, 2008. ISBN 3-900051-11-9.

Duncan Temple Lang. Using XML for statistics: The XML package. *R News*, 1(1):24–27, January 2001. URL `http://CRAN.R-project.org/doc/Rnews/`.

Duncan Temple Lang. *Sxslt: R extension for libxslt*, 2007. URL `http://www.omegahat.org/Sxslt, http://www.omegahat.org`. R package version 0.7-0.