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 a HTML report.

Contents

Chapter 1

Introduction

There has been much work into the concept of Quality Control for software[cite SWEBOK (ch11) – how?]. 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 the transformation from log file to report.

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.* As graphics output is being compared, this control output has to be initially verified as being correct, so for the rest of this report it 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. (Ihaka and Gentleman, 1996) and (R Foundation for Statistical Computing, 2008) and (Murrell and Hornik, 2003).

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 run 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 run 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 pre-defined 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 Paul Murrell (2003). It was 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 perfomed on the plots and a difference plot produced which gives a visual representation of the differences between the plots, however this is not supported on 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 record information relating to the plots, such as which plots were created in which directory, which call created the plots, and so on. Another necessary feature is 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. The current **graphicsQC** package is a complete re-write of this 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 model 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, including any warnings or errors which may have occurred. These functions are used for creation of both the model and test output and are further described in Chapter 2.

Once the model 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)[change cite to R extensions?]. 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()` is used for this. `writeReport()` generates a HTML page of the results of any plotting or comparison result. It is further discussed in Chapter 4.

Chapter 2

Plotting Arbitrary Expressions

2.1 Plotting expressions

In order to test the correctness of output, it follows that output must have initially been produced. For this task, 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 related information (such as operating system, date, directory, R version and so on). It does this by initially error checking it's arguments, and then making a call to a function within the namespace, `evalPlotCode()`.

`evalPlotCode()` is reponsible 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. Warnings however are not considered to be stop-worthy, so evaluation will continue after recording the warning. 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 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 due to the underlying code used to produce the respective graphics, or whether there is a problem or change with a specific graphics driver.

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 which 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 which 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 which 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 which 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. `path` is the path to place the plots and log in. `prefix` is the prefix to use for the files, and `clear` specifies whether to first remove files which already exist with a name like any which might be created. The resulting object is given the class `qcPlotExprResult`.

An example is given below.

```
> first <- plotExpr(c("y <- 10", "x <- 1", "plot(x:y)", "z <- 5"),
             c("pdf", "png"), "exampleDir", "firstExample")
> first
plotExpr Result:
Call:          Sweave("report.Rnw")
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
```

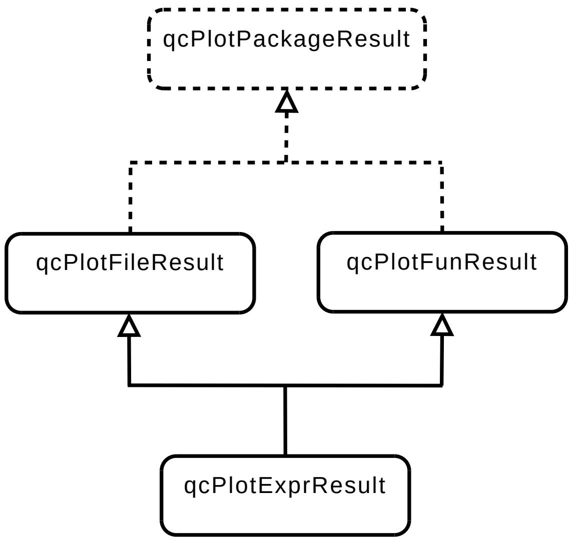

Figure 2.1: Classing structure for 'plot' objects

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 a generated by a single call to `example()` of each function.

The two functions which 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.

There is now some intrinsic classing hierachy as `qcPlotFileResult` objects and `qcPlotFunResult` objects both contain `qcPlotExprResult` objects, plus some extra information, so these objects can be considered as super-classes to `qcPlotExprResult`. Conceptually, this is easily extended to the ability to plot entire packages, as they would consist of multiple sets of `qcPlotFileResult` and `qcPlotFunResult` objects, and thus `qcPlotPackageResult` would be a super-class of these. 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. This 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, c("pdf", "ps", "png"), "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 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 which 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 create XML documents in R, the **XML** package has been used (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 it's 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 the names of them 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>Sun Nov  9 23:25:10 2008</date>
  <call>
   <![CDATA[
   Sweave("report.Rnw")   ]]>
  </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 it's 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 represent 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>Sun Nov  9 23:25:11 2008</date>
  <call>
   <![CDATA[
   Sweave("report.Rnw")   ]]>
  </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</qcPl
</qcPlotFunResult>
```

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

- GNU diff/ImageMagick are used. Diff plots only made when a diff is detected, however not made when they are going to be erased afterwards (although erase doesn't fully work yet)?

3.1 Comparing sets of expressions

3.1.1 Text Based vs. Bitmap Formats

- explanation. 3 file formats currently supported (mention this earlier?). can use xor to create plots for bitmaps? .diff files only make sense for text based. advantages and disadvantages for both?

3.1.2 Pairing plots

- how logs are paired? unpaired.

3.2 Comparing sets of files and functions

3.3 Auto-detection of logs

- One of the features.. easier for users (especially when comparing in separate versions of R and having to find the exact file name).

3.4 Extensibility for new file formats

- use of mapply in compareType nested in lapply in compareExpr (give some gory details!?) - allows for compareNEWTYPE where the function will get automatically called with the new filetype (plus would need to include the filetype in the 'valid filetypes' list).

Chapter 4

Generating Reports

4.1 The XSL Language

- stylesheet language for xml. Define templates of how to display elements.

4.2 Transforming XML logs into HTML reports

- logs are in xml, xslt seems the perfect choice (remember, this option is left open for other users). some default style sheets are provided, but are easily overwritten!

.. uses Lang (2007).

Chapter 5

A real example: grid

5.1 Plotting functions in the grid package

- ls(”package:grid”) (of course, don’t show everything)

5.2 Comparing the grid functions

- maybe show the print.qcCompareExpr result for one of the functions (even though it doesn’t fully work..)?

5.3 Reporting on the differences in grid

- print screen from firefox? any better way? link to example on http://graphicsqc.r-forge.r-project.org?

Chapter 6

Summary and discussion

can't supply all model output...?

- what's been done? + new package + plots stuff, compares stuff, pretty neat report - what can be improved? + plot/compare/writeReport package + print<lots> + compare(erase) + DTDs for logs + completely unpaired fun/file + mac (XML doesn't work)? windows (Sxslt not ported to windows!) few developers.

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 ??), the following command can be used:

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

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

Bibliography

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.

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.

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.