The graphicsQC package

Quality Control for Graphics in R

Stephen Gardiner

Supervised by Dr. Paul Murrell

Department Of Statistics
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)?]. 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 we have an initial, correct output, we 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*. After this, 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, we can identify if there has been any change in the output. If there are no differences between the two, we can validate the test output 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 F, 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[cite hornik?]. When writing packages, there is a `R CMD check` command which will perform a multitude of quality control tests on a package. 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 is very limited in usability and functionality. It can only do regression testing on the example code in given functions in R, and only identifies which plots had differences with no further information about the differences.

Further research is therefore required in order to extend the package to be platform independent, and to identify new ways to test R code such that any arbitrary code can be tested, and more information about the differences to be reported.

1.4 Outline of the structure of graphicsqc

The design of the package mainly consists of three elements. First, expressions which may (or may not) produce plots need to be evaluated. Secondly, there The main

processes involved in the **graphicsQC** package are; evaluating sets of expressions which may (or may not) produce graphical output, comparing the plots produced by the expressions

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. Using error handling functions in R, `plotExpr` was designed to be able to produce the output from any arbitrary code passed to it as an argument, with the ability to note and safely recover from any warnings and to also correctly stop on any errors.

- uses lapply over filetype (set up so warnings keep running and blocks stop at the correct time. This leads on to how the log is written).

=====unedited old stuff==== As arbitrary code is accepted by the function it can be easily seen that plotting the code from files, functions, or even packages in R is possible, and thus the convenience functions plotFile, and plotFun were also created.

The output produced from a plotting function would simply be a series of plots which the code might (or might not) have produced. In order to store these plots, a file format (or file formats) must be chosen and given as an argument to the function. 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.
For example, we might wish to create plots of the `plot` function. In R:

```
> plotFunction("plot", filetype=c("png", "pdf"), path="plotControl")
```

Then to view the files created in the directory:

```
$ ls plotControl/
plot-1.pdf  plot-2.pdf  plot-3.pdf  plot-4.pdf  plot-funLog.xml
plot-1.png  plot-2.png  plot-3.png  plot-4.png  plot-log.xml
```

As we can see, files from the `plot` function's example code have been produced, where each file has the prefix of the function name, and a numbered suffix to identify it.

Also, as specified, pdf and png files have been produced, along with two xml files; `plot-log.xml` and `plot-funLog.xml`. The creation of the XML files is described in the next section.

2.2 Plotting files and functions

- multiple calls to plotExpr.

2.3 Logs of plots created

- logs had to be made, hooks don't guarantee a (new) plot?, have to search for plots after they're made

2.3.1 The XML Language

- extensible, self-documenting, platform independent etc.

As we will want to compare the plots to previously produced plots, we will need to know information about each set of plots which has been produced. Information about the plots produced are stored in an e**X**tensible **M**arkup **L**anguage file. The main advantages of storing this information in XML is that it is self-documenting and also platform independent.

2.3.2 XML logging

- show how it's done (split info and main?) - and plotFile and plotFun since they've now been explained?. process can now be taken out of R. freedom of choice.

It is relatively human-legible, and since it is self-documenting, the process at this point can be taken out of R and the plots and corresponding information used by other programs if desired.
Continuing the example in 3.1, the log file produced is shown:

```
$ cat plotControl/plot-log.xml
<?xml version="1.0"?>
<qcPlotExprResult>
 <info>
  <OS>unix</OS>
  <Rver>2.6.1</Rver>
  <date>Sat Feb 23 11:32:49 2008</date>
  <call>
   <![CDATA[
   plotFunction("plot", filetype = c("png", "pdf"), path = "plotControl")   ]]>
  </call>
  <filetype>png</filetype>
  <filetype>pdf</filetype>
  <directory>/home/stephen/plotControl</directory>
 </info>
```

```
 <warnings/>
 <errors/>
 <filenames>
  <filename>plot-1.pdf</filename>
  <filename>plot-1.png</filename>
  <filename>plot-2.pdf</filename>
  <filename>plot-2.png</filename>
  <filename>plot-3.pdf</filename>
  <filename>plot-3.png</filename>
  <filename>plot-4.pdf</filename>
  <filename>plot-4.png</filename>
 </filenames>
</qcPlotExprResult>
```

As XML is self-documenting, the log file produced does not require much explanation. The `funLog` file is simply a pointer to all of the log files that the function might have produced for ease of comparison later.

2.4 Example: Plotting functions in the `grid` package

- ls("package:grid") (of course, don't show everything, cut some off and leave ...)

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 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?

3.2 Comparing sets of expressions

3.2.1 Pairing plots

- how logs are paired? unpaired.

The next step in the process is to compare a set of plots to another; typically between two different versions of R to establish if the graphics outputs have changed, but could also be used to detect whether changes made to the graphics system or to specific functions are different to previously known correct control group outputs.

GNU diff is used.

To detect the differences, the utility `diff` is used, which is assumed on all supported platforms according to the R Coding Standards (Writing R Extensions). An extension to this has been to use the ImageMagick software to also create plots of the differences when available. A drawback to this is that the ImageMagick software must first be installed, but it is open-source, free, and is readily available for many operating systems or even from source. While it is a distinct advantage to use the ImageMagick software, it is also not a requirement as the differences will still be detected by `diff`, but plots of the differences will not be created.
For example, to create two plots which will be different from each other:

```
> control <- plotExpr("hist(rep(1:10,1:10), breaks=6)", "png", "histControl")
> test <- plotExpr("hist(rep(10:1,1:10), breaks=6)", "png", "histTest")
```

```
> compare(test, control, erase="none")
```

Which informs us that the plots are indeed different:

```
/home/stephen/histControl-1.png
                     "different"
```

And an appropriate plot of the differences is also produced, with the differences highlighted in red:

While this was a trivial example of a single expression, the ability to compare example code from entire functions has also been implemented, with lists of the identical and different files and corresponding plots of the differences produced and also the ability to handle extreme cases such as when new plots have been added in the test group which don't have a corresponding pair in the control group.

3.3 Comparing sets of files and functions

3.4 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).

3.5 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).

3.6 Example: Comparing the `grid` functions

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

Chapter 4

Reporting

4.1 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!

4.1.1 The XSL Language

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

4.2 Example: 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 5

Conclusions

- 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.

A powerful method of doing regression testing for graphics has been implemented in R. The current method consists of generating sets of plots for the control and test groups, and then comparing them. It is a major improvement on earlier work as it is now platform independent, able to run any arbitrary code, and able to produce plots of differences. It significantly improves the reliability and quality of graphics testing in R.

The current implementation still has much room for improvement. Using the current implementation as a base, the package could be extended to include a function to plot and compare entire packages, which will simply be an extension of the current functions which plot and compare other functions. A function to write a report (such as an HTML report) based on the comparisons would also be a useful addition to the package. Lastly a wrapper class to combine plotting, comparing, and reporting into one step would be a clear finalisation of the package.

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...

References

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: Writing R Extensions. R Foundation for Statistical Computing, Vienna, Austria, 2008.