Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003)
March 20–22, Vienna, Austria *ISSN 1609-395X*
Kurt Hornik, Friedrich Leisch & Achim Zeileis (eds.)
http://www.ci.tuwien.ac.at/Conferences/DSC-2003/

Quality Assurance for Graphics in R

Paul Murrell **Kurt Hornik**

Abstract

An important part of software development involves testing the reliability of the software. R has good tools for checking that the software runs without catastrophic failure, but provides less support for checking that the software produces the correct output, particularly for graphics code. This paper discusses the sort of testing that would be desirable, several ways that this testing could be performed, and introduces some early attempts at providing tools for performing such testing.

1 Introduction

This article is concerned with the *quality* of graphics software related to the R language and environment for statistical computing and graphics (Ihaka and Gentleman, 1996).

The quality of software involves many different characteristics, such as reliability, efficiency, usability and so on. For this article, we will focus on the reliability aspect of software: whether the software is error-free and whether the software produces correct results.

1.1 Measuring software quality: QA, QC and regression tests

There are two important concepts involved in measuring the quality of software[1].

Quality Control (QC), or *testing*, involves running the software to make sure that it does not produce errors (*crash*), and to make sure that it produces the correct result. QC is concerned with the *detection* of problems (*bugs*).

A specific form of testing, called *regression testing*, involves checking that a change in the software has not caused a change in the output of the software. In

[1]Defining and measuring quality is an area rich in terminology; the definitions in this article are based on Hower (2003), Schulmeyer and McManus (1999), and Jones (1997)

this sort of testing, a set of *control output* is created prior to making a change to the software, and this is compared to a set of *test output* created after the software has been changed.

Testing that the software is producing correct output will be referred to as *validation*. As in regression testing, we have two sets of output, test and control, but we have the additional constraint that the control output has been verified to be correct. In this case, we will refer to the control output as *model output*.

Quality Assurance (QA) is concerned with the quality of the *process* that produces the software. This involves establishing and enforcing standards and procedures for software development. QA is concerned with the *prevention* of bugs.

1.2 QA and QC in R

There are a number of QC tools and QA procedures for R (Hornik, 2002).

As part of the standard documentation for R functions, it is possible to provide example code to demonstrate the correct usage of the function. In a "source" distribution of R, it is possible to run all of this example code, for all functions, via the (terminal) command `make check`. This provides a test that the examples all run without producing a crash. This command also performs some regression tests to check that certain known numerical results are produced correctly.

A similar command, `R CMD check` is available for running tests on add-on packages for R. This also performs checks on the existence and accuracy of documentation, and on its consistency with the R code.

In terms of QA, add-on packages must successfully pass the `R CMD check` tests before they are made available on the main distribution site[2]. Also, changes by the core developers of R must (at least) pass `make check-devel` tests (a more stringent set of tests than `make check`) before they are incorporated into the official development version.

1.3 QC for graphics in R

Some of the example code that is run by `make check` and `R CMD check` produces PostScript output. This means that some of the graphics code within R is being tested to make sure that it does not produce a crash, but there are several ways in which this testing is inadequate.

In order to understand some of the problems with this testing, it is necessary to look at the software involved with producing and viewing R graphics output (see Figure 1). The central piece of software is R's graphics engine – a body of C code that is distributed with R. There is R code and C code, both as part of the R distribution and written by users as extensions to R, which calls the graphics engine code. This might be referred to as "high-level" graphics code. There are a number of graphics devices, written in C code, which represent different output formats for graphics (e.g., PostScript, X11, PDF, ...); the graphics engine calls this code to produce output. Finally, there is third-party software which is required to view the different output formats. For example, ghostview may be used to view PostScript output; printer drivers and core X11 and Windows display code could be grouped in here too.

[2] http://cran.r-project.org/

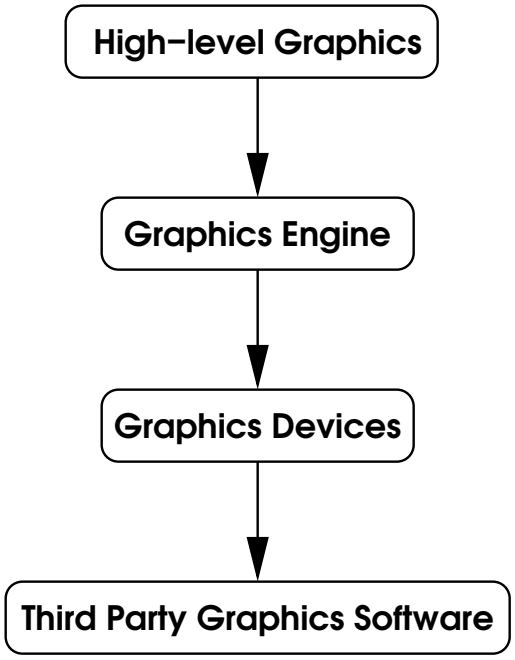

Figure 1: The structure of software associated with graphics output in R.

With this structure in mind, it is more correct to say that the current testing tools check that the high-level graphics code, the graphics engine, and the PostScript device run without crashing (for the example code). Some of the inadequacies of this testing are:

- There is no check that the output produced is correct.
- The code for all devices other than the PostScript device is not even checked to see whether it crashes.
- There is no check that the output works with third-party software. Ultimately, we require that the image that the user actually sees is correct.

This paper discusses some possible solutions to these problems.

2 Quality control tools for graphics

As mentioned, R already has tools for checking that graphics code does not crash (even if these tools are not used widely enough yet). What is still required are tools to check whether graphics output has changed (regression tests) and whether graphics output is actually correct.

2.1 Text-based versus bitmap output formats

These tools need to perform two main functions: produce output (both control output and test output) and compare sets of output.

For performing comparisons, we will use the `diff` utility which which can be assumed on all supported platforms according to the R Coding Standards (Writing R Extensions).

For producing output, we will consider both text-based and bitmap formats because neither output format on its own will satisfy all of the testing requirements.

Bitmap output is ideal for producing output which is as close as possible to what the user actually sees. It is also possible to produce bitmap output from almost all graphics devices so that all of the graphics code can be tested. Another advantage is that control and test output can be combined via an `xor` operation to produce a viewable representation of differences in output.

The downsides of bitmap output are that it has a limited resolution, produces relatively large files (especially at higher resolutions), and is very sensitive to differences in software and even hardware setup (i.e., the (version of the) third-party software used to produce the bitmap and the platform the software is running on). Another way of saying this is that the price we must pay for getting closer to what the user sees is a loss of control over the output produced (because we are going through third-party software over which we have no control). Because we are testing for *identical* files, *any* variability in output which is not due to our software changes completely compromises the testing process.

Text-based output is only possible for certain graphics devices, but it retains very high resolution, creates smaller files and is highly cross-platform. One other disadvantage with using text-based output for testing is that there are instances where the text-based output can be changed deliberately without affecting what the user sees (e.g., a change in the software for the PostScript driver which is designed only to improve the efficiency of PostScript files, or reduce their size).

2.2 Validation of graphics output

For the purposes of validation, there are two important issues. First of all, model output must not only be generated, but also verified as correct by some "expert" observer. The second issue is that the model output must be distributable – that is, we must be able to generate model output centrally then provide it to all users with the R distribution. This means that it must be both portable across platforms and not prohibitively large. Finally, a canonical set of model output must be able to be maintained by the core developers; that is, there must be one set of output which is the official set and this must be only modified or added to by an individual who is willing to vouch for the correctness of the change.

A text-based output format is the best choice in this case. Bitmap formats do not satisfy the portability requirement[3]. Also, a full set of model bitmaps for all devices at a reasonable resolution would be several hundred megabytes of files.

A text-based format restricts the number of devices we can provide model output for, but because we must create verified model output, which is time-consuming,

[3]It is possible to produce portable bitmaps in the sense that they can be sensibly transferred between systems. The problem is that it is not possible to guarantee that another system can produce exactly the same bitmap because of differences in (or even lack of) third party software.

and because the set of model output cannot be ridiculously large, we are forced to consider some sort of compromise like this in the amount of model output we provide.

Finally, text-based model output is convenient for incorporating into the existing official repository of the R distribution, which is maintained using the `cvs` version management tool.

In summary, for the purposes of providing model graphics output we cannot realistically achieve all of the desirable features outlined at the end of Section 1.3. We can provide model output to validate graphics output for a limited number of devices using a text-based output format.

2.3 Device-independent regression tests for graphics

For the simpler task of a general regression test – comparing output before a change with output after a change – we have more flexibility to achieve the goals set out at the end of Section 1.3.

It is assumed that this sort of testing takes place on a single machine (i.e., there are neither cross-platform nor distribution issues). Furthermore, we are able to ignore the strict "correctness" of output and just focus on whether change occurs in the output. This means that we are able to consider options which involve producing much larger amounts of output.

In this case, bitmap output is preferred. This allows us to test output for (almost) all devices and we are able to test what the user sees.

In summary, at the expense of dropping the strict requirement that output is tested for correctness, we are able to test more of the graphics code and test closer to what the user sees by using a bitmap output format.

2.4 Unit tests

As mentioned in Section 1.2, R already has useful tools for automatically running tests based on the example code in the documentation for R functions and objects. These tests already provide checks that a lot of R's graphics code runs without crashing. With the addition of tools to generate control output and model output, we can extend the available tools to include regression tests and validation tests of output. However, there remains an important step in making these tools effective: it is vitally important that the authors of documentation include example code for each graphical feature.

It is impossible to generate tests that will run *all* of the graphics code for *all* possible user input, but there is a serious danger of whole sections of graphics code never being tested because an example is never written to make use of that code. A good general principle to follow is always to add example code whenever a new feature is implemented. In the language of Extreme Programming (Beck, 2000), this is known as writing *unit tests*.

Another good general principle is to write example code whenever a bug is fixed – this will provide a check that the bug never resurfaces in the future.

3 Some usage scenarios

In this section we will consider the needs of various types of people who might write R graphics code.

3.1 Developers

Core R developers and authors of add-on packages will make use of both regression tests and validation tests. They will also be responsible for the generation of model output. There are two main scenarios:

developing new features: The main testing-related task here is to produce model output for any new features so that any new code is included in future testing.

bug fixing: The main weapon here is the regression test. The aim of a bug fix is usually to solve a specific problem, without adversely affecting other features which are working correctly. At the completion of the bug fix, it is probably a good idea to add a new piece of model output which can be used to ensure that the bug remains fixed after future code changes.

3.2 Patch contributors

Users who contribute patches may want to check that a patch does not adversely affect the normal performance of a function. An example is the modifications suggested by Marc Schwartz for the `barplot` function. One criterion for including the suggestions in the main R version of `barplot` is that all current examples work exactly as before. This could either be conducted as a regression test or as a validation test (or both), but is focused on a particular subset of the graphics code (as opposed to entire suites of graphics code that developers work with).

Ideally, patch contributors would also contribute example code for testing new features and/or bug fixes.

3.3 Normal users

The main use of testing for normal users is to perform a validation test on a new installation to check that the installation has succeeded. This should just be part of the standard installation checking procedure.

4 An example

In December 2002, a fix was made to the PDF device driver (part of the graphics device code). The fix involved correcting the initialisation of graphical parameters on PDF devices so differences in output were expected, but regression testing was still useful to check that the fix was having an effect in appropriate cases, and no effect elsewhere.

Control bitmap output was generated prior to the fix and test output was generated after the fix. Each set of output consisted of several hundred bitmap files. Each pair of bitmap files were then compared using the `diff` utility to detect changes.

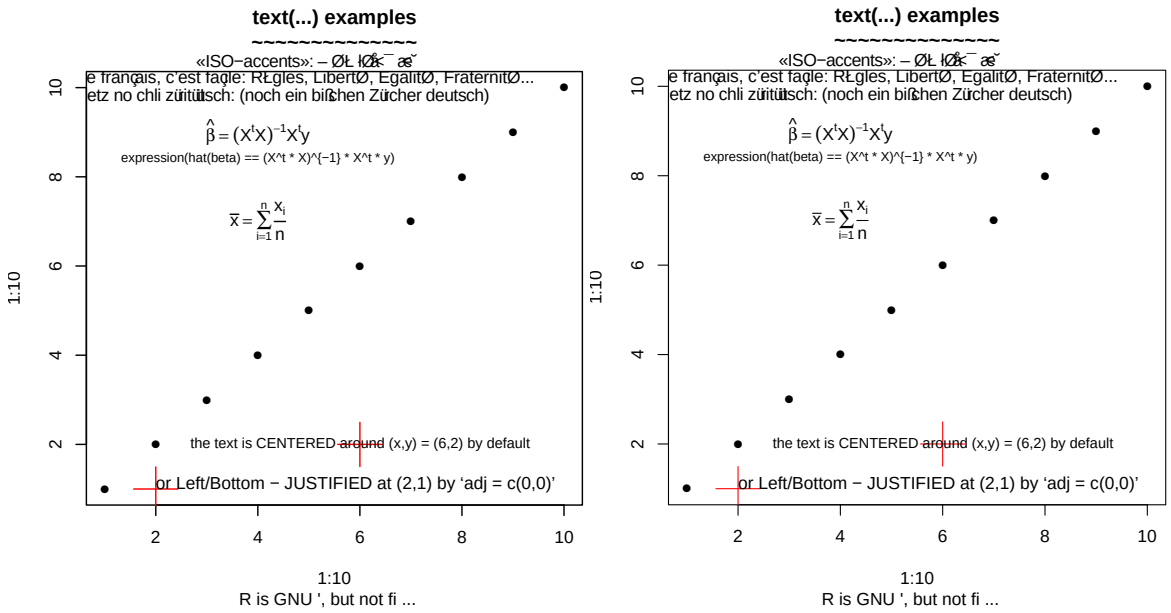

Figure 2: Control output (left) and test output (right) for detecting changes in output due to a fix in R's PDF driver.

Without automated testing tools, it is still possible to manually view different sets of output to try to detect changes. One example that was detected in the PDF driver fix demonstrates the futility and inaccuracy of such a manual approach; Figure 2 shows control and test output which differ in only two (important) pixels[4]

5 Summary

There is clearly a need for improvement in the tools available for checking graphics output from R code.

There are two main tasks that developers and users of R might want to perform: trying to detect changes in output and trying to determine whether output is correct.

We have considered two output formats for performing these tasks and neither satisfies all of the requirements for both tasks. The first task is best addressed by producing bitmap output and the second is best addressed using text-based output.

Tools are being developed for these tasks and information about an R package containing early versions of these tools is given in the Appendix.

[4]The difference is in the bar over the x in the formula for $\bar{x}$.

Some of the special characters in these figures are incorrect, but unfortunately, by the time these difficulties were noticed, it was not possible to reproduce the exact context in which the original figures were generated, so the problems have not been corrected.

References

Writing R extensions. http://cran.r-project.org/doc/manuals/R-exts.pdf.

Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.

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.

Rick Hower. Software QA and testing frequently-asked-questions. http://www.softwareqatest.com/qatfaq1.html, 2003.

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

Capers Jones. *Software Quality: Analysis and Guidelines for Success*. International Thomson Computer Press, 1997.

G. Gordon Schulmeyer and James I. McManus, editors. *Handbook of Software Quality Assurance*. Prentice Hall, third edition edition, 1999.

Corresponding author

Dr Paul Murrell
Department of Statistics
The University of Auckland
Private Bag 92019
Auckland
New Zealand
64 9 3737599 x85392
E-mail: paul@stat.auckland.ac.nz
URL: http://www.stat.auckland.ac.nz/~paul/

Appendix: The graphicsQC package

Ultimately, graphics testing tools of the sort described in this article will be incorporated into the standard QC tools distributed with R. While they are being developed, early versions are being made available as an add-on package called graphicsQC. This section provides a description of the functions that this package provides.

The package is currently available from
http://www.stat.auckland.ac.nz/~paul/.

---

graphicsQC *Quality Control Tools for Graphics*

Description

These functions provide routines for producing a set of model graphical output for specified functions, testing graphics output against the model output, and removing test and possibly model output files.

Usage

```
model.graphics(funs = NULL, package = NULL, names=NULL,
               width = 600, height = 600,
               device = postscript,
               format = "pbm",
               model.loc = ".",
               verbose = FALSE,
               reset.rng = TRUE,
               ...)

test.graphics(funs = NULL, package = NULL, names=NULL, omit=NULL,
              width = 600, height = 600,
              device = postscript, format = "pbm",
              model.loc = ".", test.loc = model.loc,
              verbose = FALSE, quiet = FALSE,
              reset.rng = TRUE,
              ...)

clean.graphics(funs = NULL, package = NULL, names = NULL,
               width = 600, height = 600,
               device = postscript, format = "pbm",
               model.loc = ".", test.loc = model.loc,
               verbose = FALSE, testonly = TRUE)
```

Arguments

funs	A character vector specifying the names of functions to produce test graphical output for.

`package`	The name of a package – all functions in the package will be tested. Only one of `funs` or `package` can be specified.
`names`	A character vector specifying the names to use as part of the filenames for the test output. This defaults to the function names. If specified, it must be the same length as the `funs` argument.
`omit`	A character vector or list. If a character vector then this specifies functions to omit. If a list then character elements specify entire functions to omit and named numeric elements specify examples within a function to omit. See the examples below.
`width`	The width of the final test output (in pixels).
`height`	The height of the final test output (in pixels).
`device`	The device to use to produce the initial test output. Valid values are: `"postscript"`, `"pdf"`, and `"x11"`.
`format`	The format for the final test output. Valid values depend on the device – some possibilities are: `"pbm"` (monochrome portable bitmap file format) for postscript, pdf and x11 devices; `"ps"` (postscript format) for the postscript and pdf device.
`model.loc`	The directory in which to look for the model output.
`test.loc`	The directory in which to create the test output.
`verbose`	If `TRUE`, print out progress messages.
`quiet`	If `TRUE`, do not print out a final result.
`testonly`	If `TRUE`, remove only test output (i.e., do not remove model output).
`reset.rng`	If `TRUE`, reset the random number generator and random seed before running each example. This will ensure that differences from examples that randomly-generate data will not show up. It is useful to set this to `FALSE` if you want to detect examples of this kind.
`...`	Additional arguments to the device.

Details

The model output is initially produced using the specified device, then converted to a final model output (as specified by the `format` argument).

The test output is initially produced using the specified device, then converted to a final test output (as specified by the `format` argument).

The test output is compared, per final output file, with the corresponding model output. If a difference is found, an image of the difference is produced (for `"bmp"` format at least).

For each function, the number of final output files, the width, height, device, and format must be the same for test and model output before the comparison will proceed.

It is possible to compare the output from functions with different names by specifying the `names` argument. This overrides the naming of output files, which is based on the function name otherwise.

Value

`model.graphics` is useful for its side-effect, which is to create an initial output file per function using the device output format (if applicable) plus a final output file per page of initial output, per function.

`test.graphics` produces a list containing a description of the differences detected. The names of the elements give the names of the functions which had examples that differed and the element values are numeric vectors indicating which examples differed. See the examples below.

WARNING

Function names which start with non-alphanumeric characters (e.g., `[`, `%`, `@`, `$`, ...), or contain `<-` are omitted).

Examples

```
model.graphics("barplot", model.loc=tempdir())
# Should detect differences due to random data generation
# in barplot examples
diffs <- test.graphics("barplot", model.loc=tempdir(), reset.rng=FALSE,
                       verbose=TRUE)
# list(barplot=1:2) as at version 1.6.1
diffs
# Should report no differences
test.graphics("barplot", omit=diffs, model.loc=tempdir())
# Tidy up
clean.graphics("barplot", model.loc=tempdir())
```