A practical guide to the R package Luminescence

Michael Dietze[1,*], Sebastian Kreutzer[2], Margret C. Fuchs[3], Christoph Burow[4], Manfred Fischer[5], Christoph Schmidt[5]

[1]Institute of Geography, TU Dresden, 01069 Dresden, Germany
[2]Department of Geography, Justus-Liebig-University Giessen, 35390 Giessen, Germany
[3]Department of Geology, TU Bergakademie Freiberg, 09599 Freiberg, Germany
[4]Institute for Geography, University of Cologne, 50923 Cologne, Germany
[5]Geographical Institute, Geomorphology, University of Bayreuth, 95440 Bayreuth, Germany

* corresponding author: micha.dietze@mailbox.tu-dresden.de

(Received 13 May 2013; in final form 14 June 2013)

Abstract
A practical guide for the **R** package 'Luminescence' is provided. An introduction on data types in **R** is given first, followed by a guideline on how to import, analyse and visualise typical SAR-OSL measurement data.

Keywords: R, luminescence dating, data analysis, plotting

Introduction

Since the **R** package 'Luminescence' has been introduced by Kreutzer et al. (2012) the developer team is continuously asked for advice from the luminescence dating community. Such requests considerably help us to further improve the package and make the tools more efficient and user friendly. However, most of these queries are not directed to specific problems of the provided functions but rather on the usage of **R** and the package in general. Motivated by an e-mail conversation with Geoff Duller this contribution aims to provide an example-based, short practical guide to **R** and the package 'Luminescence'. First, we focus on properties and ways to index different sorts of data structures, which are essential for an efficient use of the **R** package 'Luminescence'. A second section describes processing steps for luminescence data, from importing a BIN-file to plotting a D_e distribution. A third section comprises the examples in a comprehensive code section.

Throughout the manuscript **R** calls or **R** related code snippets are typed in `monospaced letters`. In some cases, numerical and graphical output was truncated for illustrative reasons.

Working with R and RStudio

R (R Development Core Team, 2013) is a freely available language and environment for statistical computing and graphics. RStudio (RStudio, 2013) is a free and open source integrated development environment (IDE) for **R**. It allows for a comfortable use of **R**.

Working with **R** usually means writing of scripts that can be executed to generate results. The fundamental advantage of working with scripts rather than clicking through graphical user interfaces or tabular calculation software is that all processed steps are formulated explicitly, i.e. every command or function call is and has to be written down. This guaranties transparent and reproducible results, easy sharing of analysis routines and flexible modification of existing approaches.

A script is a text document composed of several lines of commands, and of course explanatory comments, that can be executed by software, such as **R**. Script-based execution of command line series is much more efficient than typing of function calls into the terminal window (although this is possible).

RStudio is a comfortable "second skin" to work with **R** even more conveniently. It comprises several windows; for scripts, the command line, the workspace, plot outputs, help or a file manager. RStudio allows storing entire sessions, including the actual script and generated objects (e.g. data sets and plots), to continue working at any time.

There are a series of excellent tutorials and books about **R** (e.g. Adler, 2012; Crawley, 2012) and RStudio (e.g. Verzani, 2011) that cannot be discussed here. However, on the official website of the **R** package 'Luminescence' (http://www.r-luminescence.de) there are plenty of suggestions and

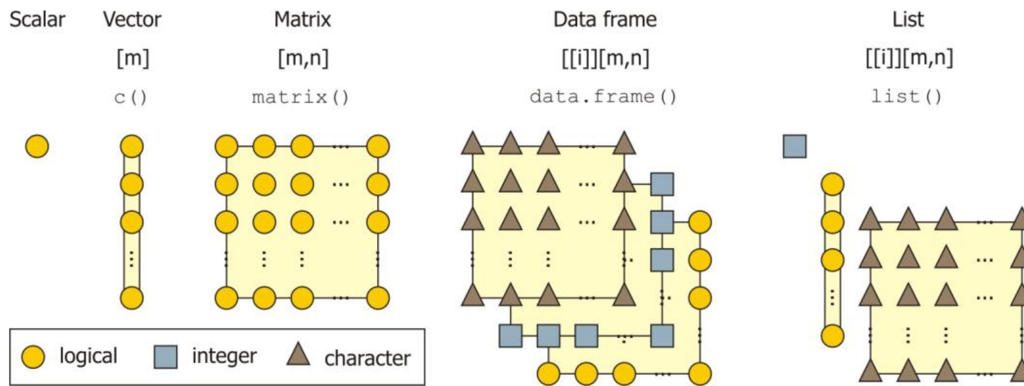

Figure 1. *Data structures in **R**, commonly used in the package 'Luminescence'. The colour and the shape of individual objects indicate similar data types (e.g. logical, integer, character) whereas their alignment represents the structure. Code under each structure definition corresponds to the creation of the structures in **R**. From left to right structures increase in complexity: scalar, vector, matrix, data frame, list. For further data structures and information cf. Crawley (2012).*

some tutorials dedicated to the use of **R** for luminescence data analysis.

Data types and structures in R

Data can be of various type. Common data types are logical (i.e. `TRUE, FALSE`), integer (e.g. `1, 2`), double (e.g. `1.2, 2.3`), complex (e.g. `2+3i, 1.3+3.2i`) and character (e.g. `"a", "b"`). There are more data types in **R** but these are of minor relevance here. The type of data determines which operations are possible (or meaningful) with this data. To infer the data type of a variable use the function `typeof()`.

Regardless of their type, data always shows a certain structure, which defines how values are organised and may be addressed. For convenient usage data may be stored in variables (or more generally in objects). It is of crucial importance to note that one variable must not necessarily comprise only one but can contain millions of individual values. **R** allows for checking the data structure of a variable with the function `str()`. To actually work with the data, it is necessary to "recall" the content of a variable, or parts of it. This is referred to as indexing. The following structures are commonly encountered when working with **R** and should therefore be introduced here. Fig. 1 shows illustrative sketches of the data structures.

Scalars: Scalars are the most simple data structure. One variable represents precisely one value (1,1 structure). Scalars can therefore be described as zero-dimensional data structures. In **R**, scalars are in fact vectors of length one. The command `x <- 1` assigns the value 1 to the variable `x`. A scalar is indexed simply by calling the variable name.

Vectors: Vectors are different from scalars in that they comprise more than one value. They contain m rows of values, organised in one column (m,1 structure). Hence, vectors can be described as one-dimensional data structures. Vectors may contain any data type but this must be consistent throughout. To infer the number of elements, the length of a vector, use the function `length()`. To index an element of a vector, its position in the vector must be specified in angular brackets after the variable name: `x[m]`. To index more than one element use either a sequence (`x[1:5]`) or a concatenation of values (`x[c(1, 2, 3, 4, 5)]`).

Matrices: Adding a further dimension yields a matrix structure. Matrices contain m rows and n columns of data (m,n structure). Hence, matrices can be described as two-dimensional data structures. Matrices can be of any, consistent data type. Indexing matrix elements requires row- and column-numbers of the target elements in angular brackets: `X[m,n]`. To index an entire row or column, just skip the respective index value: `X[1,]` or `X[,1]`.

Data frames: Data frames consist of components with the same geometry (same length of vectors or matrix rows and columns) but may contain different data types. Data frames are the most common data structure in **R**, as many functions require data frames as input arguments. Indexing elements of a data frame is a two-step task. First, the component and then the element of the respective component must be

indexed. The component is expressed by two nested angular brackets (`[[ ]]`). So indexing one element of a vector in a data frame may be similar to `dataframe[[1]][8]`.

Alternatively, the components of a data frame can be named. If names are present, the operator `$` can be used for indexing as well. For example, if there is a data frame (`dataframe`) comprising two vectors (`data` and `metadata`), one may index the first element of `metadata` by typing:

```
dataframe$metadata[1]
```

or

```
dataframe[[2]][1].
```

Lists: Similar to data frames, but also deregulating the constraint of consistent geometry and data types, lists allow handling different types and structures of data. Lists are therefore the most flexible - but not necessarily the most appropriate - data structure. Indexing follows the same rules as for data frames.

S4-objects: S4 objects are of fundamentally different data structure. They are related to object-oriented programming but may be tentatively compared to lists. They can contain several components, stored in so called slots. Details on S4-objects may be not relevant in this context. Components of S4-objects are indexed by the operator `@`. Apart from this difference, indexing is quite similar to that of data frames. Note: Although the **R** package 'Luminescence' already utilises S4-objects (e.g. `Risoe.BINfileData-class`) and the upcoming package version later this year will considerably benefit from the usage of S4-objects, details on S4-objects are not relevant for this tutorial.

From BIN-files to D_e-distributions

Prerequisites for analysing luminescence data

To work with the **R** package 'Luminescence' it is first of all necessary to install the package from CRAN; either via command line (`install.packages("Luminescence", dependencies = TRUE)`) or in RStudio via menu Tools > Install Packages. Note that the checkbox "Install dependencies" should be selected. To actually use the functionalities of the package, it must be loaded at the beginning of each **R** session. Furthermore the working directory should be set. It is good practice to load the library (i.e. the functions part of a package) and define the working directory at the beginning of a script.

```
> ## load the library
> library("Luminescence")

> ## set the working directory
> setwd("/analysis/project_0815")
```

Import and inspect BIN-files

In general, analysis of luminescence data will start with importing a BIN-file to the **R** workspace. The package provides the function `readBIN2R()` to import BIN-files from typical luminescence measurements. It creates an S4-object with two slots: `METADATA` (a data frame) and `DATA` (a list). `METADATA` contains meta-information for all measurements and is primarily used to select measurements (stored in `DATA`) based on e.g. sample position. Once imported, calling the variable displays a short summary of the object.

```
> ## import the BIN-file
> SAR.data <-
+ readBIN2R("example.BIN")

> ## show a short summary
> SAR.data

> Risoe.BINfileData Object
>  Version:          03
>  Object Date:      060120
>  User:             krb
>  System ID:        30
>  Overall Records:  600
>  Records Type:     IRSL  (n = 20)
                     OSL   (n = 340)
                     TL    (n = 220)
>  Position Range:   1 : 20
>  Run Range:        1 : 44
>  Set Range:        1 : 2
```

The example data set (example.BIN) resulted from a standard SAR protocol, applied to a sample of fluvial quartz (coarse grains, 90-160 µm) from the Pamir Plateau, analysed at TU Bergakademie Freiberg in 2013, and can be downloaded from the Ancient TL website. To create a more elaborated overview, the data frame `METADATA` must be indexed by specifying the desired columns. To show, as an example, the parameters ID (1), SEL (2), LTYPE (7), POSITION (17), RUN (18), DTYPE (23) and IRR_TIME (24) for the first five measurements, the respective column-numbers must be known (see below). In practice this includes indexing the slot `METADATA` of the S4-object `SAR.data` and then indexing the first five rows and respective columns therein:

```
> SAR.data@METADATA[1:5, c(1, 2, 7,
+ 17, 18, 23, 24)]
>   ID  SEL LTYPE POSITION RUN
+ DTYPE IRR_TIME
> 1 1 TRUE  TL  1 1 Natural     0
> 2 2 TRUE OSL  1 2 Natural     0
> 3 3 TRUE  TL  1 4 Natural     0
> 4 4 TRUE OSL  1 5 Bleach+dose 80
> 5 5 TRUE  TL  1 7 Bleach+dose 0
```

If this summary content is used frequently, it may be useful to store the column-numbers in a separate variable (`summary.01 <- c(1, 2, 7, 17, 18, 23, 24)`) for convenient use later on (`SAR.data@METADATA[,summary.01]`). This way, different summary templates can be created. A complete list of column-numbers can be displayed by `cbind(1:length(SAR.data@METADATA), colnames(SAR.data@METADATA))`.

Analyse SAR-data

Currently, the package is focused on the analysis of measurements following the SAR protocol (Murray & Wintle, 2000). The function `Analyse_SAR.OSLdata()` returns a set of parameters from individual measurement cycles in order to determine background- and sensitivity-corrected signals that may be used for growth curve estimation (see below). The function requires information about the sample (i.e. position) to be analysed, the signal integral and the background integral, along with a sample ID. By default the function `Analyse_SAR.OSLdata()` creates a graphical output for visual inspection of measurement curves (one composite plot for each position). However, for further analysis the numeric output is more important. The following example shows how to set the necessary parameters, perform an SAR analysis and what the numerical output looks like.

```
> ## define analysis parameters
> signal <- 1:5
> backgrd <- 200:250
> position <- 1:2
> info <- "Arbitrary sample 1"

> ## analyse position 1 to 2
> SAR.results <-
+ Analyse_SAR.OSLdata(
+   input.data = SAR.data,
+   signal.integral = signal,
+   background.integral = backgrd,
+   position = position,
+   info.measurement = info)

> ## display the output
> str(SAR.results)
```

The created object (`SAR.results`) is a list with three components: `LnLxTnTx`, `Rejection Criteria` and `SARParameters`, each of them composed of further objects. To access them, just move through the data structure step by step. For example if you are interested in the second cut heat temperature type `SAR.results$SARParameters$cutheat[2]`. Most important (and most complex) is the `LnLxTnTx`-list. Since two positions were analysed (`position <- 1:2`) the list contains two data frames. Each data frame consists of the number of measurements according to the applied SAR protocol. Each measurement yielded 15 parameters (such as `Name`, `Dose`, `Repeated`, `LnLx` and so on). To access the LnLx data from measurement 1 (natural dose) of position 1 type `SAR.results$LnLxTnTx[[1]]$LxTx[1]`.

Create growth curves and estimate D_e-values

From the large output amount of `Analyse_SAR.OSLdata()` the most important data sets for subsequent analyses are `Dose`, `LxTx`, `LxTx.Error` and `TnTx`. To create growth curves and estimate equivalent doses, these are needed in a data frame structure. The following code shows how to manage these steps.

```
> ## create data frame
> data.LxTx <- as.data.frame(cbind(
+   SAR.results$LnLxTnTx[[1]][2],
+   SAR.results$LnLxTnTx[[1]][12],
+   SAR.results$LnLxTnTx[[1]][13],
+   SAR.results$LnLxTnTx[[1]][6]))

> ## show the results
> data.LxTx

>   Dose       LxTx LxTx.Error TnTx
> 1    0  5.8947468 0.28838345 1862
> 2 1000  5.3317223 0.32684141 2006
> 3 1800  7.8098997 0.36604484 2239
> 4 2200  9.5146256 0.47587953 2393
> 5 3000 10.4157443 0.60718256 2891
> 6    0  0.5314526 0.07193097 2045
> 7 1800  7.1563381 0.46570722 2829
```

The function `plot_GrowthCurve()` creates a dose response curve from the measurement data. The uncertainty related to equivalent dose estimation is based on Monte Carlo simulations. The function returns the actual D_e-value, its associated error and the fit object.

```
> ## create dose response curve
> growth.curve <- plot_GrowthCurve(
+   data.LxTx)
```

```
> ## show fit parameters
> growth.curve$Fit

> ## assign De and De.error
> De.data <- cbind(
+   growth.curve$De[1:2])
```

For routine analysis it may be convenient to run this D_e modelling process in a loop for all samples of a data set.

```
> ## define analysis parameters
> signal <- 1:5
> backgrd <- 200:250
> position <- 1:20

> ## analyse positions 1 to 20
> SAR.results <-
+ Analyse_SAR.OSLdata(
+   input.data = SAR.data,
+   signal.integral = signal,
+   background.integral = backgrd,
+   position = position)

> ## Define output variable
> De.data <- data.frame(
+   De = NA,
+   De.Error = NA)

> ## Compute De values in a loop
> for(i in 1:max(position)) {
+   data.LxTx <- as.data.frame(
+     cbind(SAR.results[[1]][[i]]
+     [c(2, 12, 13, 6)]))
+   curve <- plot_GrowthCurve(
+     data.LxTx)

> ## assign De value and De error
>   De.data[i,] <- as.numeric(
+     curve$De[1:2])
+ }
```

Convert seconds to Gray

To convert the absorbed dose from seconds to the SI unit Gray the function `Second2Gray()` can be used. It includes error propagation, by default with the Gaussian approach.

```
> De.data <- Second2Gray(
+   values = De.data,
+   dose_rate = c(0.0881, 0.0006),
+   method = "gaussian")
```

Display D_e-values

There are several methods to visualise D_e distributions. Perhaps the most common ones are histograms, probability density functions based on kernel density estimates (KDE) and the radial plot (Galbraith, 1988). The chapter above illustrated how to obtain numeric data for plot outputs. One mandatory preparation step is to remove missing values (`NA`) from the `De` and `De.Error` data. This is easily done with `De.data <- De.data[complete.cases(De.data),]`.

A histogram with standard error overlay, rugs and statistical summary (Fig. 2A) can be created with the function `plot_Histogram()`.

```
> plot_Histogram(
+   values = De.data,
+   summary = c("n", "mean",
+     "median", "kdemax", "sdrel",
+     "sdabs", "serel", "seabs"))
```

Plotting a probability density plot (Fig. 2B) can be done with the function `plot_KDE()`. Further statistical summary data can be added. The following example shows most of these statistical parameters. It is left to the user to decide which parameters allow for a meaningful interpretation.

```
> plot_KDE(
+   values = De.data,
+   distribution.parameters =
+    c("mean", "median", "kdemax"),
+   summary = c("n", "mean",
+     "median", "kdemax", "sdrel",
+     "sdabs", "serel", "seabs"),
+   xlim = c(0, 450))
```

A radial plot (Fig. 2C) is created with the function `plot_RadialPlot()`. This function also supports grouped data plots, if a list with group indices is provided. For example, to plot values < 130 Gy as one group and values >= 130 Gy as a second group, the following code is needed:

```
> group.indices <- list(
+   which(De.data[,1] < 130),
+   which(De.data[,1] >= 130))
> plot_RadialPlot(
+   sample = De.data,
+   zscale.log = TRUE,
+   sample.groups = group.indices)
```

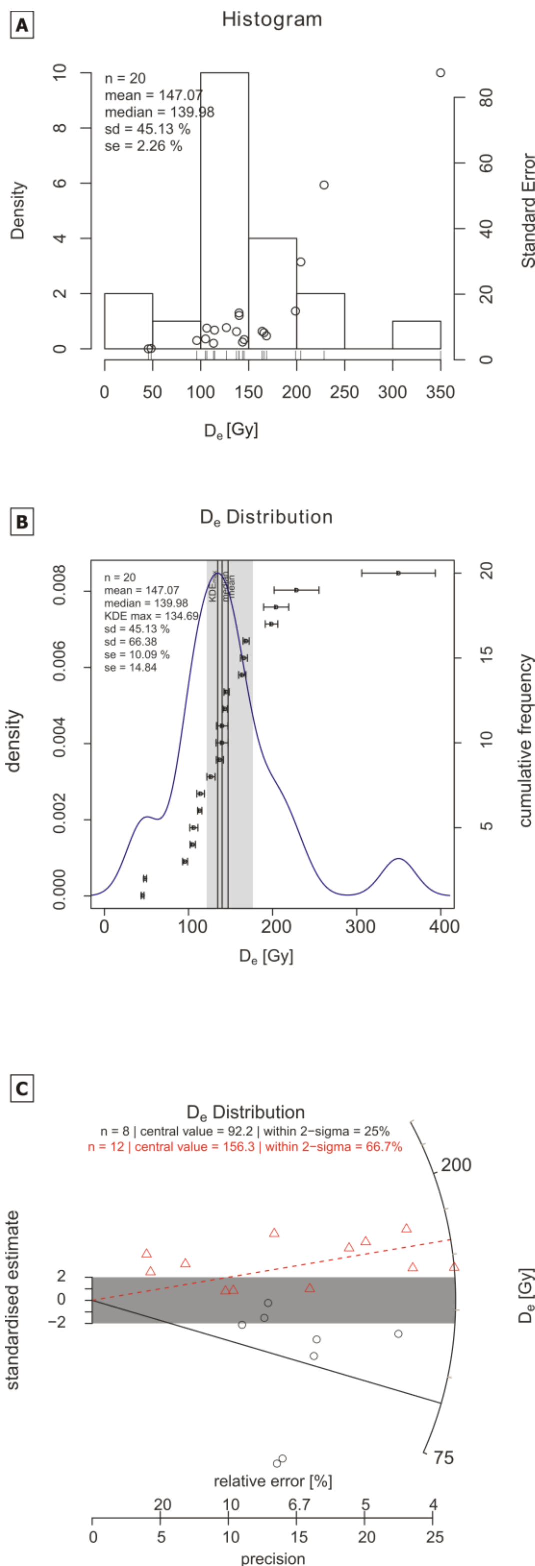

Figure 2: *Examples of plot outputs. A: histogram with rugs, standard errors and statistical measures, B: KDE-based probability density function with statistical measures, C: radial plot of grouped values.*

Save the data

R saves data in a binary format (`*.Rdata`) with the function `save()`. To save tabular data as ASCII-files use the function `write.table()`. Re-reading data is performed by `read()` or `read.table()`.

```
> ## save R-internal data
> save(SAR.data, SAR.results,
+   De.data, file =  "SAR.RData")

> ## save De data as ASCII-file
> write.table(x = De.data, file =
+ "De_data.txt", row.names = FALSE)

> ## re-read the ASCII-FILE
> De.data <- read.table(
+   "De_data.txt", header = TRUE)
```

Export graphical output

Saving graphical output when working with RStudio is quite easy. There is an export-button in the plots-window that allows for choosing from different formats and resolutions. However, it is also possible to export a plot directly using **R** commands. **R** can plot graphics to at least the following devices: `bmp()`, `jpeg()`, `png()`, `tiff()`, `pdf()`, `postscript()`, `win.metafile()`. Depending on the device, there are additional arguments such as `filename`, `width`, `height`, `pointsize`, `res`. Unless one wants to create further file output, it is important to close the respective device after a plot has been created. This is done by the function `dev.off()`. The function `graphics.off()` closes all open devices. To save for example a radial plot as jpg-file of 2000 by 2000 pixels with a resolution of 300 dpi the following code is needed:

```
> ## open the graphics device jpeg
> jpeg(
+   filename = "radial_plot.jpg",
+   width = 2000,
+   height = 2000,
+   res = 300)

> ## generate the plot output
> plot_RadialPlot(De.data,
+   zscale.log = TRUE,
+   zlab = expression(paste(D[e],
+     " [s]")),
+   sample.groups = group.indices,
+   sample.col = c("royalblue",
+     "orange3"),
+   sample.pch = c(3, 4),
+   cex.global = 0.9)

## close the graphics device
> dev.off()
```

```
## load the library
library("Luminescence")

## set the working directory
setwd("/analysis/project_0815")

## definition of analysis parameters
signal.integral <- 1:5
background.integral <- 200:250
position <- 1:20

## import the BIN-file
SAR.data <- readBIN2R("example.BIN")

## analyse the dataset
SAR.results <- Analyse_SAR.OSLdata(
  input.data = SAR.data,
  signal.integral = signal.integral,
  background.integral = background.integral,
  position = position)

## extract LxTx data and create De-values
De.data <- data.frame(De = NA, De.Error = NA)
for(i in 1:max(position)) {
  data.LxTx <- as.data.frame(
  cbind(SAR.results[[1]][[i]][c(2, 12, 13, 6)]))
  growth.curve <- plot_GrowthCurve(data.LxTx)

  ## extract and show De-value and delta De
  De.data[i,] <- as.numeric(growth.curve$De[1:2])
}

## convert seconds to Gray
De.data <- Second2Gray(
  values = De.data,
  dose_rate = c(0.08812, 0.00059),
  method = "gaussian")

## show the resulting matrix
De.data
```

Table 1: *Comprehensive script for routine SAR-OSL analysis*

A comprehensive script for routine SAR-OSL analysis

The code in Table 1 is a condensed, modified version of the explanations from above. It may serve as a skeleton for readers own scripts. The user is strongly advised to thoroughly inspect all graphical and numerical output to check data consistency and measurement appropriateness. An electronic version of the entire **R** script, and the example data set used in the analyses shown here, are provided as supplements to this paper and can be found at http://www.aber.ac.uk/ancient-tl.

Summary

A practical guide for the **R** package 'Luminescence' has been provided showing the steps from importing a BIN-file to plotting a D_e distribution. Further reading, including extensive examples and detailed definitions can be found on http://www.r-luminescence.de. For further suggestions and questions the package developer team can be contacted via team@r-luminescence.de.

References

Crawley, M.J. (2012). The R Book. pp. 1080, Wiley.
Galbraith, R.F. (1988). Graphical Display of Estimates Having Differing Standard Errors. Technometrics 30: 271–281.
Kreutzer, S., Schmidt, C., Fuchs, M.C., Dietze, M., Fischer, M., Fuchs, M. (2012). Introducing an R package for luminescence dating analysis. Ancient TL 30: 1–8.
Murray, A.S., Wintle, A.G. (2000). Luminescence dating of quartz using an improved single-aliquot regenerative-dose protocol. Radiation Measurements 32: 57–73.
R Development Core Team (2013). R: A Language and Environment for Statistical Computing. http://ww.r-project.org
RStudio (2013). RStudio: Integrated development environment for R (Version 0.97.449) [Computer software]. Boston, MA. Retrieved May 09, 2013.Available from http://www.rstudio.org/
Verzani, J. (2011). Getting Started with RStudio: An Integrated Development Environment for R. pp. 92. Sebastopol, CA USA.

Reviewer

G.A.T. Duller

Reviewers' Comment

I am very grateful to the authors for putting this together. The Luminescence package that they have developed for **R** has enormous potential, and hopefully this article will encourage those who are less familiar with **R** to start to use it.