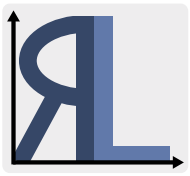
Using DRAC in R

R Luminescence Team

06 December, 2015



(package version 0.5.0 | `team@r-luminescence.de` | forum.r-luminescence.de)

Scope & Introduction

What is DRAC?

The **D**ose **R**ate and **A**ge **C**alculator (DRAC) is a web-based open access research tool to calculate environmental dose rates[1]. It is standardised, constantly updated and provides transparent calculations so that users are able to easily trace the methods used to calculate published dose rate values[2]. In this sense, DRAC is to the trapped charge dating community what CRONUS-Earth[3] is to the cosmic nuclide dating community.

> *DRAC is a Dose Rate and Age Calculator which has been designed to calculate environmental dose rates ($\dot{D}$) and ages for trapped charge dating applications. The calculations are applicable to both optically stimulated luminescence (OSL) and thermoluminescence (TL) dating and may also be useful in some electron spin resonance (ESR) applications. DRAC provides a standardised $\dot{D}$ calculator with transparent calculation using published input variables. It is an effective means of removing the potential for miscalculation, allowing improved assessment of $\dot{D}$ calculations and simpler inter-laboratory $\dot{D}$ comparisons.* - www.aber.ac.uk

DRAC has been designed and developed in equal parts by Julie Durcan and Georgina King and details of the calculation process can be found in the accompanying journal article in Quaternary Geochronology (Durcan et al., 2015).

[1] DRAC is freely accessible at `http://www.aber.ac.uk/alrl/drac`

[2] Source code on GitHub: `https://github.com/DRAC-calculator/DRAC-calculator`

[3] CRONUS-Earth online calculators: `http://hess.ess.washington.edu/`

Why use DRAC in R?

DRAC is to the dose rate what the **R** package 'Luminescence'[4](Kreutzer et al., 2012) is to the equivalent dose, the two integral variables in the age equation[5] in trapped charge dating. Combining both tools, they make a perfect match for a comprehensive, transparent and scalable analyses of chronometric data. By integrating an interface to DRAC in the 'Luminescence' package users are able to process and analyse their data without the need to switch programs. A common workflow in **R** to calculate the equivalent dose (Fuchs et al., 2015) can now be extended to calculate the dose rate and ultimately the age. As both

[4] 'Luminescence' on CRAN: `http://cran.rstudio.com/package=Luminescence`

[5] $Age = \frac{Dose\ rate(\dot{D})}{Equivalent\ Dose(D_E)}$

tools are open source a transparent and reproducible analysis from raw measurement data to ready-to-publish results can be achieved. The aim is to enable users to write scripts in **R** that cover all facets of chronometric data analysis.

Requirements

Using the DRAC interface in **R** requires version $\geq$ 0.5.0 of the 'Luminescence' package. To install the latest version from CRAN you only need to run the following from an **R** console

```
install.packages("Luminescence")
```

Getting started

The following sections are very detailed, so if you are just interested in the 'how-to' you can safely skip to the **Exercise** section. For the most impatient there is a **TL;DR** section at the end.

DRAC related functions

The **R** interface to DRAC comprises two important functions, `template_DRAC()` and `use_DRAC()`. As their names suggest, `template_DRAC()` creates an **R** object that can be used as an input template and `use_DRAC()` establishes an internet connection to DRAC, submits the data and finally returns the results. Note, that **R** objects generated by `template_DRAC()` are not required to use `use_DRAC()`. The user is also able to use and submit the official spreadsheet[6]! In addition, several functions (more precisely, methods for S3 generics) were implemented to make working with DRAC objects in **R** both easier and secure. A complete list of DRAC related methods is given in the table below.

[6] Click to download DRAC Input & Output Template

Generic	Object class	Description
`as.data.frame`	`DRAC.list`	Coerce to a `data.frame`
`print`	`DRAC.list`	Print object to console
`print`	`DRAC.highlights`	Print object to console
`[<-`	`DRAC.list`	Assign value to element in `list`
`[[<-`	`DRAC.list`	Assign value to element in `list`
`$<-`	`DRAC.list`	Assign value to element in `list`

Experienced **R** users will be familiar with the listed generics. The custom `print` methods for objects of class `DRAC.list` and `DRAC.highlights` were implemented for a prettier and more informative console output. As of DRAC version 1.1 the input and output templates comprise 53 and 276 data fields, respectively, a considerable size that the default `print` method does not handle very well. The custom `print` methods provide a better structured and readable console output and also

contain additional information such as field key, short and long descriptions and allowed inputs.

Methods for the generics `[<-`, `[[<-` and `$<-` were implemented to allow data verification and to inform users in case of invalid input. By that, we can, for example, ensure that the 'De (Gy)' field (`TI:52`) only accepts numeric values or the literal 'X'. In case of fields where the user is only allowed to pick between options provided by DRAC, e.g. 'Conversion factors' (`TI:4`), the user is automatically informed if the specified value is not within the possible options.

Finally, the method for `as.data.frame` is required to coerce the DRAC input template (a `list` object) to a `data.frame`. This is mainly used for internal purposes, but may also be called by users to obtain a more condensed representation of the input data.

DRAC objects

Along with the DRAC related functions two new **R** objects are introduced: `DRAC.list` and `DRAC.highlights`.[7] `DRAC.list` objects are generated by the `template_DRAC()` function, while the latter is returned by `use_DRAC()` inside the `RLum.Results` object.

[7] Actually, these are only classes attached `list` and `data.frame` objects. For the ease of understanding we will treat them as discrete objects.

DRAC Input Template - `DRAC.list`

A DRAC input template object is easily created by running the following

```
template <- template_DRAC(nrow = 1)
```

We can check the class of this object...

```
class(template)
```

```
> [1] "DRAC.list" "list"
```

...and see that this objects inherits two classes: `DRAC.list` and `list`. Note, that the class of an **R** object does not necessarily indicate its underlying data type.

```
typeof(template)
```

```
> [1] "list"
```

So `DRAC.list` objects are only a usual `list`, which **R** users are used to working with. The only difference is the attached `DRAC.list` class that is required to define custom methods for base **R** S3 generics.[8]

The structure of the `DRAC.list` objects is as follows

[8] Method dispatch for S3 generics uses the `class` attribute of the object to determine the correct method to call.

```
str(template, max.level = 0)
```

```
> List of 53
>  - attr(*, "class")= chr [1:2] "DRAC.list" "list"
```

```
str(template, list.len = 2, nchar.max = 20)
```

```
> List of 53
>  $ Project ID                                          : atomic [1:1] RLum
>   ..- attr(*, "required")= logi TRUE
>   ..- attr(*, "allowsX")= logi FALSE
>   ..- attr(*, "key")= chr "TI:1"
>   ..- attr(*, "description")= chr "Inputs can be alpha"| __truncated__
>  $ Sample ID                                           : atomic [1:1] 999
>   ..- attr(*, "required")= logi TRUE
>   ..- attr(*, "allowsX")= logi FALSE
>   ..- attr(*, "key")= chr "TI:2"
>   ..- attr(*, "description")= chr "Inputs can be alpha"| __truncated__
>   [list output truncated]
>  - attr(*, "class")= chr [1:2] "DRAC.list" "list"
```

Each DRAC input element is represented by a `list` element, while each element of the `list` has a set of attributes to provide further information about the input field.

Note, that in **R** non-standard variable names require the use of single back-slashes (grave accent, backtick). When copying code from this document make sure to replace the ticks with single back-slashes.

```
attr <- attributes(template$`Conversion factors`)
str(attr, nchar.max = 24)
```

```
> List of 6
>  $ levels     : chr [1:4] "AdamiecAitken1998" "Guerinetal2011" "Liritzisetal2013" "X"
>  $ class      : chr "factor"
>  $ required   : logi FALSE
>  $ allowsX    : logi TRUE
>  $ key        : chr "TI:4"
>  $ description: chr "The conversion factors "| __truncated__
```

All available attributes are given in the table below.

Attribute	Element	Description
`levels`	`factors`	Contains the valid input options
`class`	all	Class of the element
`required`	all	Is this field required by DRAC?
`allowsX`	all	Allow 'X' as input or not
`key`	all	The key as used by the DRAC
`description`	all	The field description from the DRAC webpage

DRAC Output Highlights - `DRAC.highlights`

After running `use_DRAC()` the user is returned an `RLum.Results` object,[9] which contains the DRAC calculation results.

[9] See help("RLum.Results-class") for reference.

```
template$'ExternalU (ppm)' <- 3
template$'errExternal U (ppm)' <- 0.1
template$'De (Gy)' <- 20
template$'errDe (Gy)' <- 0.2

DRAC.results <- use_DRAC(template)
```

Here, `DRAC.results` is an S4 object of class `RLum.Result`, which is a common data type when using the 'Luminescence' package. Of course, the usual functions can be applied to this object.

```
print(DRAC.results)
```

```
>
>  [RLum.Results]
>    originator: use_DRAC()
>    data: 4
>    .. $DRAC : list
>    .. $data : DRAC.list
>    .. $call : call
>    .. $args : list
```

```
str(DRAC.results@data, max.level = 1, nchar.max = 20)
```

```
> List of 4
>  $ DRAC:List of 6
>  $ data:List of 53
>   ..- attr(*, "class")= chr [1:2] "DRAC.list" "list"
>  $ call: language use_DRAC(template)
>  $ args:List of 1
```

We see that the object `DRAC.results` has slot `@data`, which is a `list` with four elements and one being a `list` named DRAC. Here, we will find our results. As usual, `RLum`-objects should be accessed via the `get_RLum` function.

```
results <- get_RLum(DRAC.results, "DRAC")
str(results, max.level = 1, nchar.max = 20)
```

```
> List of 6
>  $ highlights:Classes 'DRAC.highlights' and 'data.frame': 1 obs. of  25 variables:
```

```
>  $ header    : chr "<head><meta charset"| __truncated__
>  $ labels    :'data.frame':   1 obs. of  281 variables:
>   .. [list output truncated]
>  $ content   :'data.frame':   1 obs. of  281 variables:
>   .. [list output truncated]
>  $ input     :'data.frame':   1 obs. of  53 variables:
>  $ output    :'data.frame':   1 obs. of  198 variables:
>   .. [list output truncated]
```

When calling `get_RLum` we specified that the `DRAC` element should be returned and assigned to the variable `results`. The `results` object itself is again a `list` with the elements `highlights, header, labels, content, input` and `output`.

Element	Data type	Description
`highlights`	`data.frame`	summary of 25 most important input/output fields
`header`	`character`	HTTP header from the DRAC server response
`labels`	`data.frame`	descriptive headers of all input/output fields
`content`	`data.frame`	complete DRAC input/output table
`input`	`data.frame`	DRAC input table
`output`	`data.frame`	DRAC output table

From the second to last output we also saw that the `highlights` element of the list inherits the class `DRAC.highlights` (in addition to `data.frame`). Like the `DRAC.list` which was just a common `list`, `DRAC.highlights` is just a common `data.frame`. The sole purpose of this additional class is again, the possibility for custom S3 generic methods. For `DRAC.highlights` objects only a custom `print` method is provided. Apart from that, the `DRAC.highlights` object can be used just as any other `data.frame`.

```
results$highlights$'Age (ka)'
```

```
> [1] "28.623"
> attr(,"key")
> [1] "TO:GO"
```

```
results$highlights[ ,25]
```

```
> [1] "0.541"
> attr(,"key")
> [1] "TO:GP"
```

```
results$highlights["Project ID"]
```

```
> TI:1 = Project ID:
>   RLum
```

To get an overview of all fields in the `data.frame` you can do the following:

```
names(results$highlights)
```

```
>  [1] "Project ID"
>  [2] "Sample ID"
>  [3] "Mineral"
>  [4] "Water corrected alphadoserate"
>  [5] "Water corrected erralphadoserate"
>  [6] "Water corrected betadoserate"
>  [7] "Water corrected errbetadoserate"
>  [8] "Water corrected gammadoserate (Gy.ka-1)"
>  [9] "Water corrected errgammadoserate (Gy.ka-1)"
> [10] "Internal Dry alphadoserate (Gy.ka-1)"
> [11] "Internal Dry erralphadoserate (Gy.ka-1)"
> [12] "Internal Dry betadoserate (Gy.ka-1)"
> [13] "Internal Dry errbetadoserate (Gy.ka-1)"
> [14] "Cosmicdoserate (Gy.ka-1)"
> [15] "errCosmicdoserate (Gy.ka-1)"
> [16] "External doserate (Gy.ka-1)"
> [17] "External errdoserate (Gy.ka-1)"
> [18] "Internal doserate (Gy.ka-1)"
> [19] "Internal errdoserate (Gy.ka-1)"
> [20] "Environmental Dose Rate (Gy.ka-1)"
> [21] "errEnvironmental Dose Rate (Gy.ka-1)"
> [22] "De (Gy)"
> [23] "errDe (Gy)"
> [24] "Age (ka)"
> [25] "errAge (ka)"
```

Working with DRAC template objects

In the previous section we learned about the internal structure and characteristics of DRAC template objects. Here, we will focus more on how to query and assign values to that template object.

First, we will create a new DRAC template.

```
template <- template_DRAC(nrow = 1)
```

Note, that you can specify the number of input rows by using the argument nrow. You will not be able to change the number of rows after the object was created. Since the input template has 53 different fields, it is likely that you will not know all of them by heart. To get an overview you can always call print(template), for a specific element of the list you can do the following:

```
template[[1]]
```

```
> [1] "RLum"
> attr(,"required")
> [1] TRUE
> attr(,"allowsX")
> [1] FALSE
> attr(,"key")
> [1] "TI:1"
> attr(,"description")
> [1] "Inputs can be alphabetic, numeric or selected symbols (/ - () [] _). Spaces are not permitted."
```

```
template$‘Sample ID‘
```

```
> [1] "999"
> attr(,"required")
> [1] TRUE
> attr(,"allowsX")
> [1] FALSE
> attr(,"key")
> [1] "TI:2"
> attr(,"description")
> [1] "Inputs can be alphabetic, numeric or selected symbols (/ - () [] _). Spaces are not permitted."
```

To assign a new value you can use all the usual approaches.

```
template[[1]] <- "My Project ID"
template$‘Sample ID‘ <- "My Sample ID"
template[3] <- "PM"
```

As mentioned earlier, the DRAC.list methods for the S3 generics [<-, [[<-, $<- are to ensure data validity. The following examples will fail for various reasons.

```
template$‘Project ID‘ <- 42
```

```
> Warning: Project ID : Input must be of class character
```

```
template$‘ExternalU (ppm)‘ <- "a number"
```

```
> Warning: ExternalU (ppm) : Input must be of class numeric
```

```
template$Mineral <- "Quartz"
```

```
> Warning: Mineral : Invalid option. Valid options are: Q, F, PM
```

```
template$‘De (Gy)‘ <- c(10, 20)
```

```
> Warning: De (Gy) : Input must be of length 1
```

Once all input data is provided you can quickly evaluate the input template by coercing the `DRAC.list` to a data.frame.[10]

[10] Do not worry, the table looks nicer in your **R** console.

```
as.data.frame(template)
```

```
>            TI:1         TI:2 TI:3             TI:4 TI:5 TI:6 TI:7 TI:8
> 1 My Project ID My Sample ID   PM Liritzisetal2013    0    0    0    0
>   TI:9 TI:10 TI:11 TI:12 TI:13 TI:14 TI:15 TI:16 TI:17 TI:18 TI:19 TI:20
> 1    0     0     0     0     Y     0     0     0     0     0     0     0
>   TI:21 TI:22 TI:23 TI:24 TI:25 TI:26 TI:27 TI:28 TI:29 TI:30 TI:31 TI:32
> 1     0     Y     0     0     0     0     0     0     0     0     Y   100
>   TI:33           TI:34            TI:35 TI:36 TI:37    TI:38 TI:39 TI:40
> 1   150 Brennanetal1991 Guerinetal2012-Q     8    10 Bell1979     0     0
>   TI:41 TI:42 TI:43 TI:44 TI:45 TI:46 TI:47 TI:48 TI:49 TI:50 TI:51 TI:52
> 1     0     0     X     X     X     X     X     X     X     X     X     X
>   TI:53
> 1     X
```

What might be particularly useful when writing scripts is the `blueprint` argument for `print`. This will print a “blueprint” of the template to the console that can be copy-pasted to a script. So instead of writing down all assignments manually, you call the following line once and copy all the lines to your script.

```
print(template, blueprint = TRUE)
```

```
> template$‘Project ID‘ <- c(’My Project ID’)
> template$‘Sample ID‘ <- c(’My Sample ID’)
> template$‘Mineral‘ <- c(’PM’) # OPTIONS: Q, F, PM
```

```
> template$‘Conversion factors‘ <- c(’Liritzisetal2013’) # OPTIONS: AdamiecAitken1998, Guerinetal2011, Lir
> template$‘ExternalU (ppm)‘ <- c(0)
> template$‘errExternal U (ppm)‘ <- c(0)
> template$‘External Th (ppm)‘ <- c(0)
> template$‘errExternal Th (ppm)‘ <- c(0)
> template$‘External K (%)‘ <- c(0)
> template$‘errExternal K (%)‘ <- c(0)
> template$‘External Rb (ppm)‘ <- c(0)
> template$‘errExternal Rb (ppm)‘ <- c(0)
> template$‘Calculate external Rb from K conc?‘ <- c(’Y’) # OPTIONS: Y, N
> template$‘Internal U (ppm)‘ <- c(0)
> template$‘errInternal U (ppm)‘ <- c(0)
> template$‘Internal Th (ppm)‘ <- c(0)
> template$‘errInternal Th (ppm)‘ <- c(0)
> template$‘Internal K (%)‘ <- c(0)
> template$‘errInternal K (%)‘ <- c(0)
> template$‘Rb (ppm)‘ <- c(0)
> template$‘errRb (ppm)‘ <- c(0)
> template$‘Calculate internal Rb from K conc?‘ <- c(’Y’) # OPTIONS: Y, N, X
> template$‘User external alphadoserate (Gy.ka-1)‘ <- c(0)
> template$‘errUser external alphadoserate (Gy.ka-1)‘ <- c(0)
> template$‘User external betadoserate (Gy.ka-1)‘ <- c(0)
> template$‘errUser external betadoserate (Gy.ka-1)‘ <- c(0)
> template$‘User external gamma doserate (Gy.ka-1)‘ <- c(0)
> template$‘errUser external gammadoserate (Gy.ka-1)‘ <- c(0)
> template$‘User internal doserate (Gy.ka-1)‘ <- c(0)
> template$‘errUser internal doserate (Gy.ka-1)‘ <- c(0)
> template$‘Scale gammadoserate at shallow depths?‘ <- c(’Y’) # OPTIONS: Y, N
> template$‘Grain size min (microns)‘ <- c(100)
> template$‘Grain size max (microns)‘ <- c(150)
> template$‘alpha-Grain size attenuation‘ <- c(’Brennanetal1991’) # OPTIONS: Bell1980, Brennanetal1991
> template$‘beta-Grain size attenuation ‘ <- c(’Guerinetal2012-Q’) # OPTIONS: Mejdahl1979, Brennan2003, Gu
> template$‘Etch depth min (microns)‘ <- c(8)
> template$‘Etch depth max (microns)‘ <- c(10)
> template$‘beta-Etch depth attenuation factor‘ <- c(’Bell1979’) # OPTIONS: Bell1979, Brennan2003, X
> template$‘a-value‘ <- c(0)
> template$‘erra-value‘ <- c(0)
> template$‘Water content ((wet weight - dry weight)/dry weight) %‘ <- c(0)
> template$‘errWater content %‘ <- c(0)
> template$‘Depth (m)‘ <- c(’X’)
> template$‘errDepth (m)‘ <- c(’X’)
> template$‘Overburden density (g cm-3)‘ <- c(’X’)
> template$‘errOverburden density (g cm-3)‘ <- c(’X’)
> template$‘Latitude (decimal degrees)‘ <- c(’X’)
```

```
> template$‘Longitude (decimal degrees)‘ <- c(’X’)
> template$‘Altitude (m)‘ <- c(’X’)
> template$‘User cosmicdoserate (Gy.ka-1)‘ <- c(’X’)
> template$‘errUser cosmicdoserate (Gy.ka-1)‘ <- c(’X’)
> template$‘De (Gy)‘ <- c(’X’)
> template$‘errDe (Gy)‘ <- c(’X’)

>
>    You can copy all lines above to your script and fill in the data.
```

This automatically creates assignment statements for all input fields with the correct number of values and, in case of factors, adds the valid options as comments.

Using the DRAC input spreadsheet

For the `use_DRAC()` function the user is **not** required to work with the **R** objects created by the `template_DRAC()` function. You can always download the official DRAC input & output template[11], fill in the data and provide the full path for the argument `file` when using `use_DRAC()`. The function will then import and parse the spreadsheet to create the necessary input string.

[11] Click to download DRAC Input & Output Template

```
pathToFile <- "~/DRAC_Input_and_Output_Template.xlsx"
use_DRAC(file = pathToFile)
```

Exercise: Example data from DRAC

DRAC provides an examplary data set comprising dose rate information for a quartz, feldspar and polymineralic sample. As a worked example we will reproduce this data set in **R** using the template object generated by `template_DRAC()`. First, we create a template with three rows.

```
TI <- template_DRAC(nrow = 3)
```

Now we have to fill in the data. Note that all fields in the template come with a default value, so you are not always required to fill in data for each input field. The template itself, however, will not work without data being input. To save us a lot of typing we then call `print(input, blueprint = TRUE)` in the **R** console, copy all lines to the script and start filling in the data.

```
TI$‘Project ID‘ <-
  c(’DRAC-example’, ’DRAC-example’, ’DRAC-example’)
```

```
TI$‘Sample ID‘ <-
  c(’Quartz’, ’Feldspar’, ’Polymineral’)
TI$‘Mineral‘ <-
  c(’Q’, ’F’, ’PM’) # OPTIONS: Q, F, PM
TI$‘Conversion factors‘ <-
  c(’AdamiecAitken1998’, ’AdamiecAitken1998’, ’AdamiecAitken1998’)
# AdamiecAitken1998, Guerinetal2011, Liritzisetal2013, X
TI$‘ExternalU (ppm)‘ <-
  c(3.4, 2.0, 4.0)
TI$‘errExternal U (ppm)‘ <-
  c(0.51, 0.2, .4)
TI$‘External Th (ppm)‘ <-
  c(14.47, 8.0, 12.0)
TI$‘errExternal Th (ppm)‘ <-
  c(1.69, 0.4, 0.12)
TI$‘External K (%)‘ <-
  c(1.2, 1.75, 0.83)
TI$‘errExternal K (%)‘ <-
  c(0.14, 0.05, 0.08)
TI$‘External Rb (ppm)‘ <-
  c(0, 0, 0)
TI$‘errExternal Rb (ppm)‘ <-
  c(0, 0, 0)
TI$‘Calculate external Rb from K conc?‘ <-
  c(’N’, ’Y’, ’Y’) # OPTIONS: Y, N
TI$‘Internal U (ppm)‘ <-
  c(’X’, ’X’, ’X’)
TI$‘errInternal U (ppm)‘ <-
  c(’X’, ’X’, ’X’)
TI$‘Internal Th (ppm)‘ <-
  c(’X’, ’X’, ’X’)
TI$‘errInternal Th (ppm)‘ <-
  c(’X’, ’X’, ’X’)
TI$‘Internal K (%)‘ <-
  c(’X’, 12.5, 12.5)
TI$‘errInternal K (%)‘ <-
  c(’X’, 0.5, 0.5)
TI$‘Rb (ppm)‘ <-
  c(’X’, ’X’, ’X’)
TI$‘errRb (ppm)‘ <-
  c(’X’, ’X’, ’X’)
TI$‘Calculate internal Rb from K conc?‘ <-
  c(’X’, ’N’, ’N’) # OPTIONS: Y, N, X
```

```
TI$`User external alphadoserate (Gy.ka-1)` <-
  c('X', 'X', 'X')
TI$`errUser external alphadoserate (Gy.ka-1)` <-
  c('X', 'X', 'X')
TI$`User external betadoserate (Gy.ka-1)` <-
  c('X', 'X', 2.5)
TI$`errUser external betadoserate (Gy.ka-1)` <-
  c('X', 'X', 0.15)
TI$`User external gamma doserate (Gy.ka-1)` <-
  c('X', 'X', 'X')
TI$`errUser external gammadoserate (Gy.ka-1)` <-
  c('X', 'X', 'X')
TI$`User internal doserate (Gy.ka-1)` <-
  c('X', 'X', 'X')
TI$`errUser internal doserate (Gy.ka-1)` <-
  c('X', 'X', 'X')
TI$`Scale gammadoserate at shallow depths?` <-
  c('N', 'Y', 'Y') # OPTIONS: Y, N
TI$`Grain size min (microns)` <-
  c(90, 180, 4)
TI$`Grain size max (microns)` <-
  c(125, 212, 11)
TI$`alpha-Grain size attenuation` <-
  c('Brennanetal1991', 'Bell1980', 'Bell1980')
# OPTIONS: Bell1980, Brennanetal1991
TI$`beta-Grain size attenuation ` <-
  c('Guerinetal2012-Q', 'Mejdahl1979', 'Mejdahl1979')
# Mejdahl1979, Brennan2003, Guerinetal2012-Q, Guerinetal2012-F
TI$`Etch depth min (microns)` <-
  c(8, 0, 0)
TI$`Etch depth max (microns)` <-
  c(10, 0, 0)
TI$`beta-Etch depth attenuation factor` <-
  c('Bell1979', 'Bell1979', 'X')
# OPTIONS: Bell1979, Brennan2003, X
TI$`a-value` <-
  c(0, 0.15, 0.086)
TI$`erra-value` <-
  c(0, 0.05, 0.0038)
TI$`Water content ((wet weight - dry weight)/dry weight) %` <-
  c(5, 10, 10)
TI$`errWater content %` <-
  c(2, 3, 5)
```

```
TI$'Depth (m)' <-
  c(2.2, 0.15, 'X')
TI$'errDepth (m)' <-
  c(0.22, 0.015, 'X')
TI$'Overburden density (g cm-3)' <-
  c(1.8, 1.8, 'X')
TI$'errOverburden density (g cm-3)' <-
  c(0.1, 0.1, 'X')
TI$'Latitude (decimal degrees)' <-
  c(30.0, 60.0, 'X')
TI$'Longitude (decimal degrees)' <-
  c(70.0, 100.0, 'X')
TI$'Altitude (m)' <-
  c(150, 200, 'X')
TI$'User cosmicdoserate (Gy.ka-1)' <-
  c('X', 'X', 0.2)
TI$'errUser cosmicdoserate (Gy.ka-1)' <-
  c('X', 'X', 0.1)
TI$'De (Gy)' <-
  c(20, 15, 204.47)
TI$'errDe (Gy)' <-
  c(0.2, 1.5, 2.69)
```

Before we submit the data we have a last check by coercing the template to a `data.frame`.[12]

[12] For better readability the shown output table was generated with the `xtable` package.

```
as.data.frame(TI)
```

	TI:1	TI:2	TI:3	TI:4	TI:5	TI:6	TI:7	TI:8	TI:9	TI:10	TI:11	TI:12	TI:13	TI:14	TI:15
1	DRAC-example	Quartz	Q	AdamiecAitken1998	3.40	0.51	14.47	1.69	1.20	0.14	0.00	0.00	N	X	X
2	DRAC-example	Feldspar	F	AdamiecAitken1998	2.00	0.20	8.00	0.40	1.75	0.05	0.00	0.00	Y	X	X
3	DRAC-example	Polymineral	PM	AdamiecAitken1998	4.00	0.40	12.00	0.12	0.83	0.08	0.00	0.00	Y	X	X

	TI:16	TI:17	TI:18	TI:19	TI:20	TI:21	TI:22	TI:23	TI:24	TI:25	TI:26	TI:27	TI:28	TI:29	TI:30
1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
2	X	X	12.5	0.5	X	X	N	X	X	X	X	X	X	X	X
3	X	X	12.5	0.5	X	X	N	X	X	2.5	0.15	X	X	X	X

	TI:31	TI:32	TI:33	TI:34	TI:35	TI:36	TI:37	TI:38	TI:39	TI:40	TI:41	TI:42	TI:43	TI:44	TI:45
1	N	90.00	125.00	Brennanetal1991	Guerinetal2012-Q	8.00	10.00	Bell1979	0.00	0.00	5.00	2.00	2.2	0.22	1.8
2	Y	180.00	212.00	Bell1980	Mejdahl1979	0.00	0.00	Bell1979	0.15	0.05	10.00	3.00	0.15	0.015	1.8
3	Y	4.00	11.00	Bell1980	Mejdahl1979	0.00	0.00	X	0.09	0.00	10.00	5.00	X	X	X

When we have verified the data we can submit it to DRAC. `use_DRAC()` transforms the data into a submission string and sends an HTTP request to the DRAC web-page.

	TI:46	TI:47	TI:48	TI:49	TI:50	TI:51	TI:52	TI:53
1	0.1	30	70	150	X	X	20	0.2
2	0.1	60	100	200	X	X	15	1.5
3	X	X	X	X	0.2	0.1	204.47	2.69

```
TO <- use_DRAC(file = TI, name = "RLum")
```

In case of a successful request the HTTP response from the DRAC web-page will be parsed and saved in **R** objects. Along with the original input data all results, including the HTTP header, will be returned as an `RLum.Results` object. If you are only interested in the DRAC output "highlights" you can run

```
highlights <- get_RLum(TO)
```

Alternatively, in `get_RLum` you can specify to return all DRAC results by

```
results <- get_RLum(TO, "DRAC")
```

Here, the `results` object is a `list` of length six with the elements `highlights, header, labels, content, input` and `output` (see previous sections for reference). From here on, you are free to do with the data whatever you like. Here, we will only check if reproducing the example data was succesful. The expected age estimates for the quartz, feldspar and polymineral samples are: 6.702 ± 0.374 *ka*, 4.22 ± 0.442 *ka*, 41.765 ± 2.241 *ka*.

```
string <- paste0("\n",
                 highlights$'Sample ID', ": ",
                 highlights$'Age (ka)', " \U00B1 ",
                 highlights$'errAge (ka)', " ka")
cat(string)
```

```
>
> Quartz: 6.702 ± 0.374 ka
> Feldspar: 4.22 ± 0.442 ka
> Polymineral: 41.765 ± 2.241 ka
```

TL;DR

- A perfect match: DRAC is open-source and transparent, so is the **R** package 'Luminescence'
- DRAC calculates $\dot{D}$, 'Luminescence' the D_E. Hence...
  - $Age = \frac{D_E}{\dot{D}} = \frac{'Luminescence'}{DRAC}$

- The **R** DRAC interface uses two functions:
  - `template_DRAC()` creates an input template object
  - `use_DRAC()` sends the data to DRAC
- New methods for S3 generics `[<-`, `[[<-`, `$<-` ensure data validity[13]
- How to use:
  1. Create a template:
     - `t <- template_DRAC()`
  2. Fill in data:
     - t[[1]] <- "My Project"
     - t$External K (%) <- 12.5
  3. Too many fields to memorise?
     - Use `print(t, blueprint = TRUE)`
     - Copy all lines from console to your script
  4. Send data to DRAC:
     - res <- use_DRAC(t)
  5. Access the results:
     - h <- get_RLum(res)
  6. Marvel at the results:
     - print(paste(h$Age (ka), "±", h$errAge (ka)))
- Always make sure you cite the use of DRAC in your work, published or otherwise (Durcan et al., 2015)

[13] Slight exaggeration, there is no definite type safety in **R**. There are probably thousands of ways to overwhelm the data validation, but such is the beauty of **R**.

Acknowledgements

DRAC has been designed and developed in equal parts by Julie Durcan and Georgina King.

References

Durcan, J.A., King, G.E., Duller, G.A.T., 2015. DRAC: Dose rate and age calculation for trapped charge dating. Quaternary Geochronology 28, 54–61. doi:10.1016/j.quageo.2015.03.012

Fuchs, M.C., Kreutzer, S., Burow, C., Dietze, M., Fischer, M., Schmidt, C., Fuchs, M., 2015. Data processing in luminescence dating analysis: An exemplary workflow using the R package Luminescence. Quaternary International 362, 8–13. doi:10.1016/j.quaint.2014.06.034

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.