Guerrilla Automation with R

*R automation despite resource constraints*

**Introduction**

Guerrilla warfare is a type of asymmetric warfare wherein a smaller, less powerful military uses unconventional tactics to fight a stronger, more traditional military. It often involves a higher level of mobility, with nimblness making up for the military’s lack of resources. We can draw an analogy in business. When a problem calls for a traditional technological solution, but no such solution is currently in-place, actuaries may require a low-cost, nimble approach to meeting the needs of their stakeholders.

That’s where R comes in. R is a free and open source programming language with a powerful and continually-expanding set of libraries and packages. It is highly versatile and compatible with a wide variety of other systems. I am going to walk through a hypothetical use-case for R that highlights R’s advantages. Example code will be interspersed throughout the article. The full code, complete with example excel files can be accessed via Github at the following url:

<https://github.com/TimothyQuast/Guerrilla_Automation>

If you download the repository and would like to follow along, I highly recommend installing RStudio for free from [Rstudio.com](https://Rstudio.com). A rudimentary understanding of R will be helpful in following along, but it is not essential.

**The Use-Case**

Let’s suppose that our stakeholder has several ledger account balances they want to support using the actuarial workpapers that feed them. They want to break each balance into pieces, with each piece corresponding to one actuarial workpaper which contributes to that balance. The actuarial workpapers are stored in Excel in a regular format in different places throughout the stakeholder’s network drives, with a variety of teams contributing to the same account balances. Moreover, the workpapers represent an aggregated version of the actuarial results, so they aren’t overly granular. But they are granular enough to make a manual solution infeasible. How do we get the data we need to support the account balances?

The proper, traditional method is a big fancy subledger containing the supporting balances along with metadata that traces each balance. But let’s say the stakeholder doesn’t have a big fancy subledger yet. Further, let’s say that the stakeholder wants the balances supported *soon* – sooner than a big fancy subledger can be developed. We need a temporary solution to solve the problem quickly. In such a conundrum, one might try automating the task with R!

R is suitable for the task for several reasons:

* **It’s free!** It can be used for such a task without requiring investment dollars or licensing.
* **It’s open source with a broad user base.** Developers are continually producing marvelous new packages that can solve tough problems, making R extremely versatile.
* **It’s highly compatible.** The variety of packages and the non-proprietary nature of the system make it uniquely capable of interacting with other systems, such as Excel.
* **It’s suitable for rapidly prototype.** R is elegant and fairly high-level, allowing the user to do a lot with just a little code.

R has some disadvantages too:

* **It’s less efficient.** R doesn’t do as well as other languages in terms of processing efficiency. That’s why it’s important that the workpapers are already aggregated. If they were extremely granular, with a high volumne of data, then R might struggle.
* **There’s a learning curve.** Like all programming languages, R must be learned. It might be difficult for new personnel to learn the language, making the process less portable.

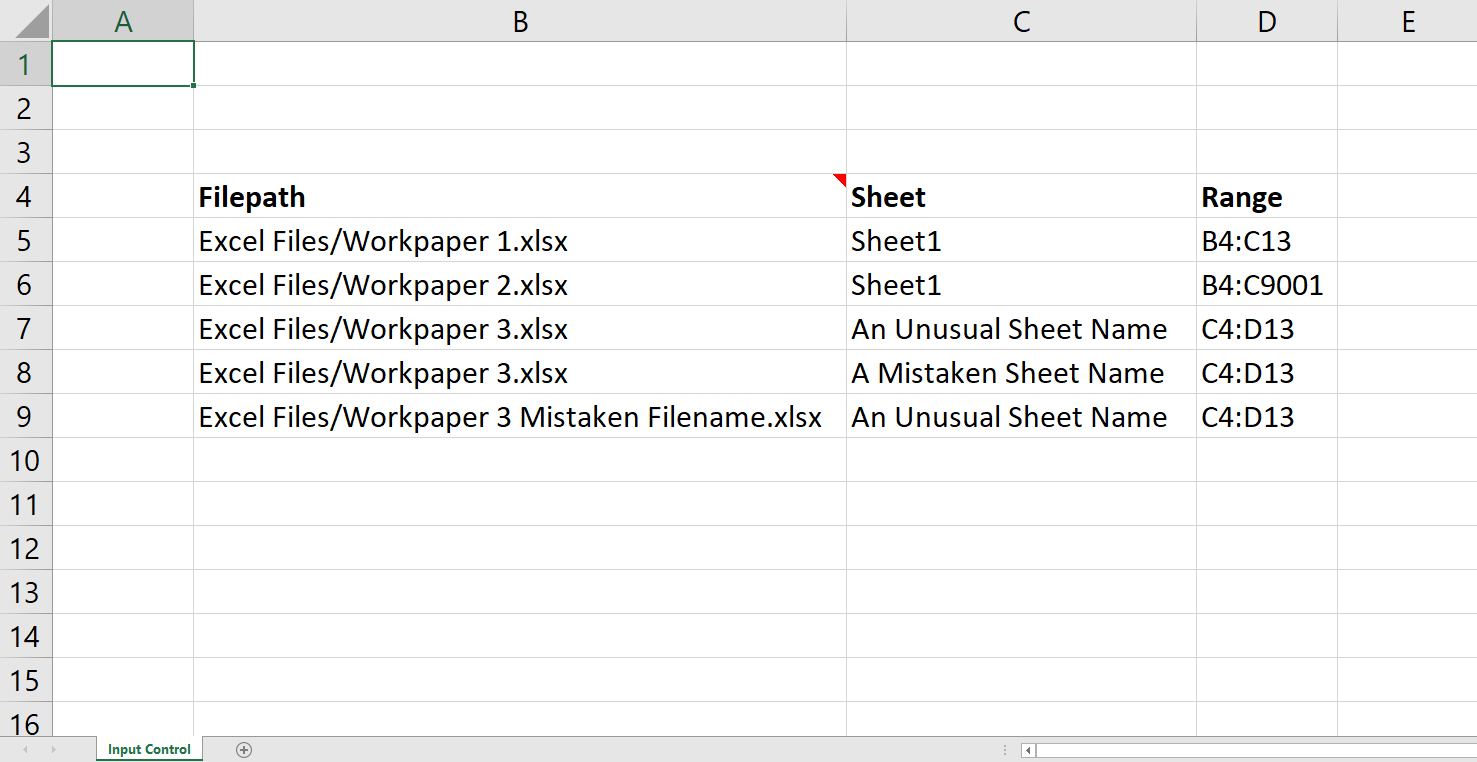
The drawbacks are not overwhelming and it won’t cost us anything but time to try, so let’s figure out how to solve our problem with R. I’ve constructed an example problem (which you can find in the GitHub repository) using 4 Excel files. **Ledger Balances.xlsx** contains the “ledger balances” we are trying to support. Three workpaper files contain the “workpapers” that support the “ledger balances”. Pretend that the workpaper files are located in disparate places throughout the stakeholder’s network drives!

|  |  |
| --- | --- |
| **Excel Files/Ledger Balances.xlsx** | **Excel Files/Workpaper 1.xlsx** |
| **Excel Files/Workpaper 2.xlsx** | **Excel Files/Workpaper 3.xlsx** |

You can manually verify that the account number totals in the 3 workpapers sum to the ledger balance totals, but then what’s the point of automating it? Notice that the workpapers keep the data in a regular format, but **Workpaper 3.xlsx** has the data in different cells and it has an unusual sheet name! These differences are intentional, and we will handle them in our solution.

**The Solution**

The first step to solving any problem is simple: get organized! I start by chronicling the input files in the **Excel Files/Input Control.xlsx** spreadsheet.



I typically prefer to use absolute file paths, but I set this up with relative paths so that it will work on other machines. The first three rows here contain correct information. The last 2 contain errors in the sheet name and file name respectively. This can happen if, for example, the process is run on a monthly basis and the file-naming conventions for a workpaper change from one month to the next. I included them to demonstrate R’s ability to deal with these complications. Two things to note:

* The range for Workpaper 2 is unnecessarily large. This is a useful tactic when the numbers of rows changes from month to month and you want to make sure you capture all the data. We eventually remove the resulting empty rows.
* There are no errors in the Range column. A runtime error in the Range column would be captured in the same way as file/sheet name errors, but a logical error (i.e. if we entered the range incorrectly) could create challenges. It is possible to dynamically determine the correct range for each sheet, but doing so is a bit trickier, and beyond the scope of this article.

Next, we start using R. We rely on three packages: readxl (Wickham & Bryan 2018), writexl (Ooms, 2018), and dplyr (Wickham et al., 2020). If you have Rstudio installed, you can follow along in the GitHub repository by opening the file **Guerrilla Automation.Rproj** and then opening the file **Guerrilla Automation.R** from the same instance of Rstudio. You can run the whole process from start to finish by calling either the **main** or the **output\_results** functions. Note that you can install the requisite packages using the following code. You also need to source the R script file:

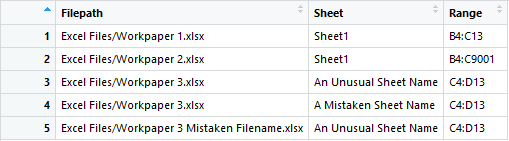
install.packages(c("readxl","writexl", "dplyr"))

source('Guerrilla Automation.R')

Step 1 is to import the Input Control data. We do this in the **import\_input\_contro**l function with the following lines:

input\_control = read\_excel(filepath, sheet = "Input Control", range="B4:D9001")  
input\_control = data.frame(input\_control[!is.na(input\_control$Filepath),])   
  
 Note that the range parameter is excessively large. This helps if you have to add new files to the input control. The second line converts input\_control to a data frame and removes the NA values at the end of the data (which occur because of the excessive range). The input\_control data frame now looks as follows:

**input\_control**



Step 2 is to loop through the input\_control data frame and use read\_excel on the parameters in each row. We do this in the **gather\_input\_data** function. We also handle our erroneous example rows in the Input Control using a tryCatch. Similarl to how we removed the extraneous rows from the Input Control, we also removed the extraneous rows from Workpaper 2. I’ve omitted the code for Step 2 because it is rather lengthy, but it results in a list of two data frames. One is an augmented version of input\_control and the other is called input\_data. I assign these data frames to input\_list via:

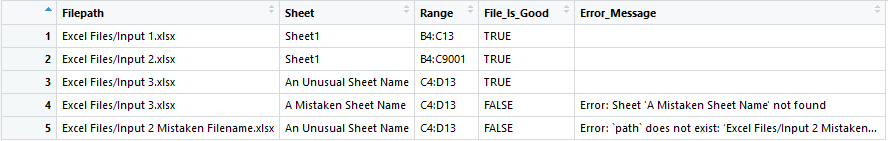
input\_list = list()

# Code omitted

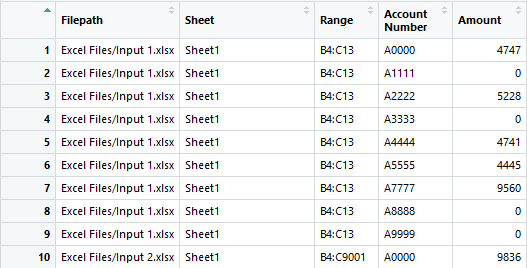
input\_list[["input\_control"]] = input\_control  
input\_list[["input\_data"]] = input\_data

Now they are as follows:

**input\_list[[“input\_control”]]**

****

**input\_list[[“input\_data”]] (rows 1-10 out of 27)**



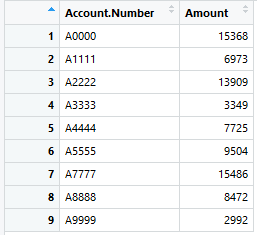
Note the two extra columns in input\_control. We can use these to track which files imported successfully and what went wrong with the files that failed.

Step 3 is to summarize the data. We do this in the **aggregate\_input\_data** function using elegant dplyr functions. We store our results in a list called aggregate\_list. We keep the old input\_list along with our new summary.

aggregate\_list = list()  
aggregate\_list[["input\_control"]] = input\_list[["input\_control"]]  
aggregate\_list[["input\_data"]] = input\_list[["input\_data"]]  
aggregate\_list[["summary"]] = data.frame(input\_list[["input\_data"]] %>% group\_by(`Account Number`) %>% summarise(Amount = sum(Amount)))

The summary now appears as follows:

**aggregate\_list[["summary"]]**



Note that these values are exactly the same as the ones we started with in **Ledger Balances.xlsx**. Huzzah!

Step 4 is to output the results. We do this in the **output\_results** function using the writexl package as follows:

write\_xlsx(aggregate\_list, "./Excel Files/R Output/Results.xlsx")

We’re back in Excel! In the file **Excel Files/R Output/Results.xlsx**, we have a tab for each data frame. I chose to output the results into a new Excel file. This prevents issues that occur when you are trying to write to a file that is currently open. If you open the **Results.xlsx** file and run the process again, you should get an error. One way to prevent this is to include a time stamp in the file name via:

write\_xlsx(aggregate\_list, paste("./Excel Files/R Output/Results ", format(Sys.time(), "%Y%m%d %s"), ".xlsx", sep=""))

Now we can easily see which files imported successfully and which didn’t and we can trace each ledger balance back to the files that contributed to it. Our approach to automating data gathering has a lot of flexibility. Here’s a few other things we could do with our solution:

1. Add additional identifier columns to **Input Control.xlsx** which can provide helpful splits to our aggregate data.
2. Add functionality that dynamically detects the appropriate range for each input file.
3. Automate the task of comparing the resulting summary to the original ledger balance.
4. Iteratively correct **Input Control.xlsx** using the **input\_control** tab in **Results.xlsx** until all of the rows contain correct information.

As you can see, R offers a lot of power and compatibility as a free and open source system. I enjoy using R in both business and as a hobby because of its elegance and versatility. It is a great resource for rapidly developing solutions that can meet the needs of your stakeholder. That’s all folks. I hope you enjoyed this “R-ticle” and found it most helpful!

**References**

Hadley Wickham and Jennifer Bryan (2018). readxl: Read Excel Files. R package version 1.1.0. https://CRAN.R-project.org/package=readxl

Jeroen Ooms (2018). writexl: Export Data Frames to Excel 'xlsx' Format. R package version 1.0.

https://CRAN.R-project.org/package=writexl

Hadley Wickham, Romain François, Lionel Henry and Kirill Müller (2020). dplyr: A Grammar of Data Manipulation. R package version 0.8.5. https://CRAN.R-project.org/package=dplyr