# Invertible Reproducible Documents

Eric Lim

October 2, 2014

# *Motivation*

**Knitr(REF)** is a wonderful package that enables dynamic report generation with R. It allows implementation of R code into LaTeX, LyX, HTML, Markdown, AsciiDoc, and reStructuredText documents through the concepts of literate programming (REF), which involves interaction between code and documentation for report generation. The main purpose of **knitr** follows an important idea in academic research that the ideal research paper or report must encompass the entire computational environment used to produce the results in the paper such as the code so that the same results can be reproduced using the same computational environment (reproducible research REF). The importance of reproducibility can be further extended to be applied in the entire area of academia (REF).

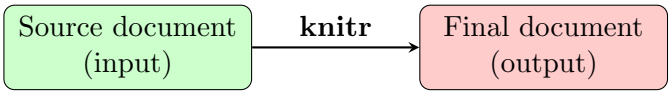


**Figure 1.1:** Unidirectional document generation by **knitr**

While there are tremendously important ideas to consider and many advantages, the process of generating R code embedded documents using **knitr** is almost always a one-way trip (Figure 1.1), meaning source documents (as input) can only generate final documents (as input), not the other way around. This is simply due to the fact that **knitr** is designed with intention to dynamically generate reports, not to extract displayed R code in final documents in order to generate source documents.

Consider a situation where a *consultant* provides reports as reproducible documents and a *client* is to read or review the reports. In this situation, it is often difficult for the consultant to receive feedback from the client efficiently. Since it is impossible to convert back from final documents to source documents, the client would have to provide his or her feedback by either writing physically on the printed copy of the final report or by electronic means such as through exchanging e-mails. The document provider, then, has to rectify the source documents accordingly, only to repeat the process of generating and presenting the report to the client. This process often has to be repeated until final correction can be achieved. As we can see, it can quickly become tedious.

We believe this is relevant to the field of statistics as similar situations

mentioned above can often arise. Interaction between clients and consultants is crucial for statisticians and any possible factor to deteriorate the relationship with clients is best avoided. Therefore, we want to avoid this issue by a more efficient document generation workflow.

A possible solution is to allow clients to interact directly with the final documents and edit them. Then consultants can merge the changes and make final adjustments to generate new reports efficiently. In order to achieve this, reports must be reproducible as well as *invertible*, that is, final reports must be able to be converted into the source document format without introducing any change from the inversion process itself.

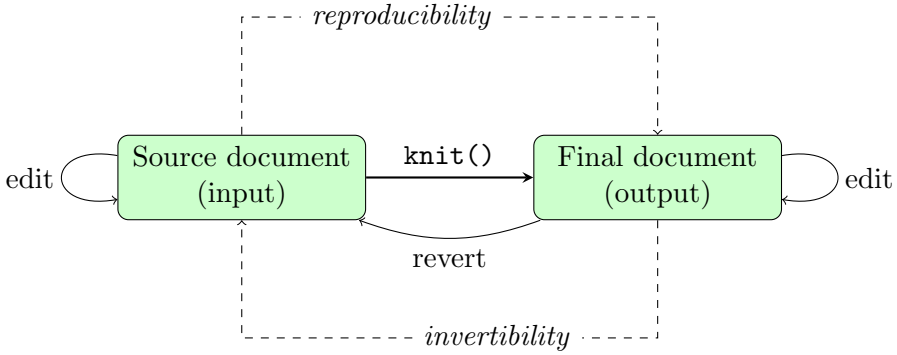


**Figure 1.2:** Invertible reproducible workflow

This report introduces a possible workflow that can be utilised effectively in the aforementioned situation. The main focus of the project is in achieving this hypothetical round trip and exploring possible issues associated with it. Generalising the workflow by improving the robustness and efficiency of the functions involved has only been considered to a level that has been manageble under the time given for the project. Hence, it has been possible to devise two separate strategies to deal *specifically* with two different document types under the allowed time. The phase 1 of this report will discuss primarily on dealing with HTML-based documents and the second phase will focus on Markdown-based documents.

# *Phase 1: R HyperText Markup Language*

## 2.1 Introduction

Dynamic report generation in the form of an HTML (REF) document requires an R HTML document as the input for **knitr**. An R HTML document contains R Code Chunks, which are lines of R code to be processed with options to customise the output. Otherwise the format of an R HTML document is essentially the same as that of a typical HTML document.

For **knitr**, this means that the source document format is the same as final document format. Using this property, difficulties in converting a document of one format into another of different format can be avoided because both the pre- and post-converted documents are of the same format.

HTML is an XML (REF)-like markup language, which means powerful XML-based tool sets, such as XPath (REF), are available for use to manipulate documents effectively. These two benefits can improve the chance of completing the round trip and the initial experiment has been decided to focus on HTML-based workflow.

Here is a brief illustration of how an HTML source document in the R HTML format can produce the final HTML document.

Listing 2.1 is the code structure of the source document, `example1.Rhtml`. We can notice from the lines 8–10 that a line of R code (line 9) is delimited by `<!--begin.rcode` and `end.rcode-->` markup.

**Listing 2.1:** `example1.Rhtml`

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4  </head>
5    <body>
6      <h1>Example</h1>
7      <p>Summary of the cars data:</p>
8      <!--begin.rcode echo=FALSE
9      summary(cars)
10     end.rcode-->
11   </body>
12 </html>
```

The following R code is used to *knit* `eample.Rhtml` to generate `example1.html`.

```
library(knitr)
knit("example1.Rhtml")
```

Listing 2.2 shows the code structure of the final document, `example1.html`. The R Code Chunk has been modified (lines 13–27) to match the HTML syntax with the CSS style markup. Depending on the *theme* specified in the source document, the `style` node will change and and these marked-up elements will be displayed accordingly.

The output from the R code is in lines 19–25 in Listing 2.2. Cautions must be taken as the internal code in Listing 2.2 has been tidied up with line endings to enhance readability. The actual untidied code consists of frequent long lines that usually require text-wrap to fit into the page.

**Listing 2.2:** (tidied) `example1.html`

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <style type="text/css"> ... </style>
5  </head>
6    <body>
7      <h1>Example</h1>
8      <p>Summary of the cars data:</p>
9      <div class="chunk" id="unnamed-chunk-1"><div class="rcode">
10       <div class="output"><pre class="knitr r">
11         ##     speed          dist
12         ##  Min.   : 4.0  Min.   :  2
13         ##  1st Qu.:12.0  1st Qu.: 26
14         ##  Median :15.0  Median : 36
15         ##  Mean   :15.4  Mean   : 43
16         ##  3rd Qu.:19.0  3rd Qu.: 56
17         ##  Max.   :25.0  Max.   :120
18       </pre></div>
19     </div></div>
20   </body>
21 </html>
```

Figure 2.1 shows what `example1.html` will look like in a web browser.

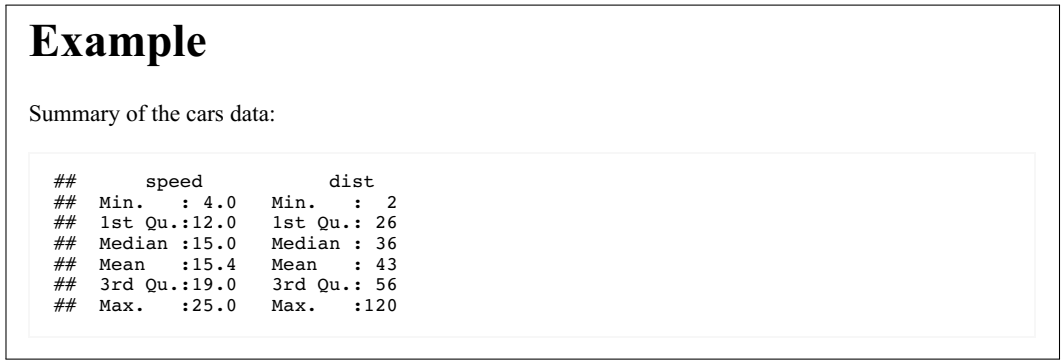


**Figure 2.1:** `example1.html` in a browser

## 2.2 Overview

A complete cycle of the invertible workflow consists of six stages. Each stage is involved primarily with text processing, along with an occasional use of XPath. The R package, **XML** (REF), is used when manipulating documents in terms of XPath.

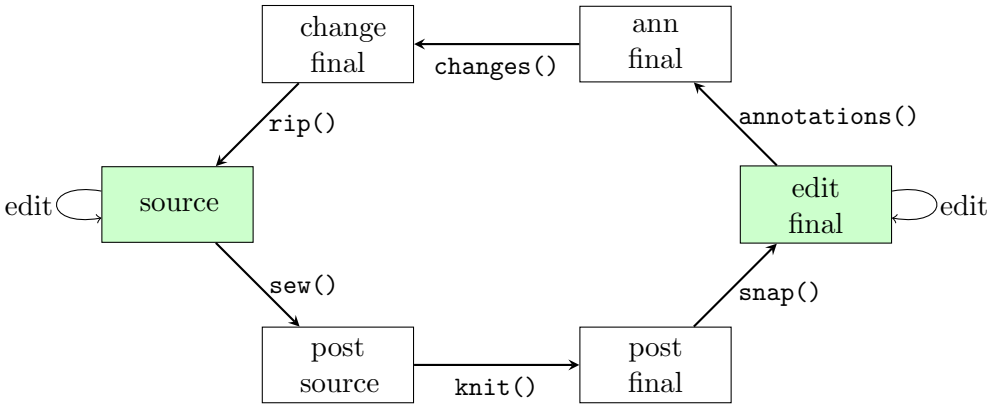


**Figure 2.2:** The reproducible, invertible workflow

### Loss of information

Processing of a source document into a final document can be associated with loss of information from the source document. For example, the original R code in line 9 of Listing 2.1 does not appear in the final document (Listing 2.2). As a solution, a source document undergoes a pre-processing step where each R Code Chunk is copied in such a way that the original R Code Chunks are perfectly preserved throughout the stages of the workflow and

are retainable after the complete cycle. The pre-processing task is carried out by the function `sew()`.

### Gain of information

On the other hand, the process of document generation can result in gain of new information. For example, lines 13–23 of Listing 2.2, which correspond to the output generated from the original R code in line 9 of Listing 2.1, only exist in post-processed final documents. The sections of newly gained infromation, i.e., the output of R Code Chunks, may appeal to the client for editing. But translating changes made on the new information from the final document to the source document where the new information is non-existent (the output is not generated yet) is problematic. And more technical side of the report, such as manipulating R code, is usually best left for the consultant. A solution chosen for this project is to restrict these R-realted sections of the final document to be annotated only. In this way, the client can still pass on feedbacks to the consultant by leaving annotations without directly changing the sections of R code and the output of the code. Annotations are allowed on the final document through a GUI in a web browser by the use of the **Annotator** (REF) JavaScript library.

### Editing final document

Allowing the client to modify the final document is achieved by adding JavaScript code to the final document. The JavaScript code loads a library called **CKEditor** (REF) that enables a word-processing GUI on the document through a browser to provide relatively familar word-processing experience to the client. Both **CKEditor** and **Annotator** are added to the final document by `snap()`.

Annotations and changes made on the final document are merged by the functions, `annotations()` and `changes()`, respectively. The final stage of the invertible workflow involves text processing to remove any artifacts added by **knitr** and is carried out by the function, `rip()`.

# *Phase 2: R Markdown*

## 3.1 Introduction

The reproducible, invertible workflow of phase 1 has limitations that could be avoided if more time is spent on it. But it is equally as important to explore whehter the workflow is actually *good* enough. Answering this question is a priorty worthwhile to explore that may help for future directions of the project. Therefore, phase 2 of the project focuses on using the workflow for other document format. The format we have chosen is Markdown (REF) for two main reasons: flexible conversion into various document formats and its increasing popularity.

Markdown (REF) is a plain text formatting syntax, with strong emphasis on readability. It can be processed by the widely used document converter, **Pandoc** (REF), into various document formats such as the popular PDF (REF) and HTML, as well as the support for conversion into a Microsoft Word or OpenOffice document. Because of this flexibility to support other formats and its native easy-to-read and -write syntax, Markdown is becoming increasingly popular.

we have decided to experiment on possibilities of devising a new workflow specifically to deal with Markdown. Rather than trying to generalise the workflow, so that it becomes robust enough to support other document formats, it is best to

The reproducible, invertible workflow is highly plausible to work on HTML-based documents, which can be converted from Markdown-based source documents using **Pandoc**. A new workflow suited specifically for Markdown will then invovle addtional steps

Similarly with R HTML, **knitr** requires a document in R Markdown (REF) format that is processed into the final document format of choice. R Code Chunks are coded in the R Markdown-based source document, in a conceptually similar way to the R HTML. With a recent update, R Markdown v2 supports integration of **Shiny** (REF) applications to generate interactive documents to further increase its popularity. With these considerations, phase 2 of this project has been decided to focus on devising a simlar reproducible, invertible workflow suited for a round trip between R Markdown and HTML.

Listing 3.1 is the internal code structure of a simple R Markdown doc-

ument, named `example2.Rmd`. R Code Chunks must be delimited by the delimiters in lines 6 and 8 of Listing 3.1 in R Markdown.

**Listing 3.1:** `example2.Rmd`

````
1 Example
2 ========================
3
4 Summary of the cars data:
5
6 ```{r}
7 summary(cars)
8 ```
````

Using **knitr**, this source document is *knitted* into Markdown which is then *rendered* to produce the final document in HTML format. Running the function, `knit()`, on the source R Markdown document only converts it into Markdown to produce a Markdown document. We can directly generate the final document by using the `render()` function in **rmarkdown** package which first calls `knit()` and then *renders* the Markdown document generated by `knit()` into the final document format. The below R code is run to generate the final document.

```
rmarkdown::render("example2.Rmd")
```

Listing 3.2 is the code structure of the final document generated from the source document. The original R Code Chunk in lines 6–8 of Listing 3.1 is processed to produce the result seen in lines 14–20 of Listing 3.2. Comparing Listings 2.2 and **??**, the two final documents are vaguely similar in structure although some of the markup used are different.

**Listing 3.2:** (tidied) `example2.html`

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <script> ... </script>
5    <style> ... </style>
6  </head>
7    <body>
8      <style type="text/css"> ... </style>
9      <div class="container-fluid main-container">
10       <div id="example" class="section level1">
11         <h1>Example</h1>
12         <p>Summary of the cars data:</p>
13         <pre><code>
```

```
14          ##     speed          dist
15          ##  Min.   : 4.0  Min.   :  2
16          ##  1st Qu.:12.0  1st Qu.: 26
17          ##  Median :15.0  Median : 36
18          ##  Mean   :15.4  Mean   : 43
19          ##  3rd Qu.:19.0  3rd Qu.: 56
20          ##  Max.   :25.0  Max.   :120
21        </code></pre>
22      </div>
23    </div>
24    <script> ...bootstrap table styles... </script>
25    <script> ... mathjax ... </script>
26  </body>
27 </html>
```

Figure 3.1 shows what `example2.html` will look like in a web browser. Figures 2.1 and 3.1 look identical.

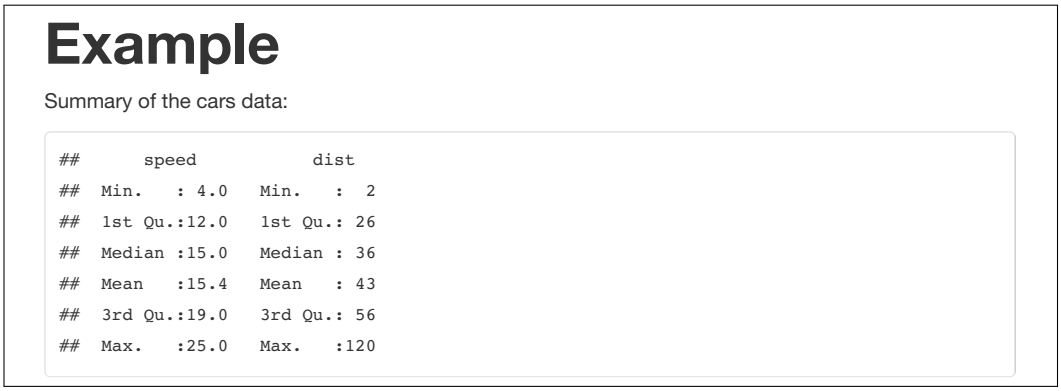


**Figure 3.1:** `example2.html` in a browser

## 3.2 Overview

(Add tikz flow chart)

### Invertibility of Pandoc

As an initiative, **Pandoc** is tested whether it supports perfect, flawless conversion between two document formats, that is, the source document must be identical to the final converted document.

Intuitively, we can use **Pandoc** on a Markdown-based source document to convert it into HTML, then back to Markdown (e.g., a round trip) and compare the source and final Markdown documents. But these two steps can be further simplified into a single step; the Markdown-based source document is converted directly to a final *Markdown* document because the output of these two workflows are conceptually the same. Hence a simple round trip on a Markdown document is tested for invertibility.

Listing 3.3 is the code structure of the source Markdown document, `example3.Rmd`.

**Listing 3.3:** `example3.Rmd`

````
1  Header 1
2  ------------------------------------------------
3  This is an R Markdown document. Markdown is a
4  simple formatting syntax for authoring web pages.
5
6  Use an asterisk mark, to provide emphasis such as
7  *italics* and **bold**.
8
9  You can write `in-line` code with a back-tick.
10
11 ```
12 Code blocks display
13 with fixed-width font
14 ```
15
16 > Blockquotes are offset
````

The command line below can be run to use **Pandoc** on the source document to generate the final document, `exmaple3-pandoc.Rmd`.

```
pandoc -t md -f md example3-return.Rmd example3.Rmd
```

Listing 3.4 is the code structure of the final document, `example3-return.Rmd`. Comparing the source and final documents, we can notice a few differences. Firstly, the number of dashes (line 2 of Listings 3.3 and 3.4) are different. Secondly, the character width for each line is changed. And finally, the fenced code syntax as seen in lines 17 and 20 of Listing 3.3 disappears and the code block is delimited by tabs (or 4 spaces) in lines 18 and 19 of Listing 3.4.

**Listing 3.4:** `example3-return.Rmd`

```
1  Header 1
2  --------
3
```

```
4  This is an R Markdown document. Markdown is a simple formatting
       syntax
5  for authoring web pages.
6
7  Use an asterisk mark, to provide emphasis such as *italics* and
8  **bold**.
9
10 You can write ‘in-line‘ code with a back-tick.
11
12     Code blocks display
13     with fixed-width font
14
15 > Blockquotes are offset
```

Evidently, **Pandoc** does not support flawless conversion between document formats and defeats the core concept of invertibility. However, further round trip using the final document, `example3-return.Rmd`, as the input produces the identical output. This suggests that **Pandoc** has an internal document structure, that is a set of rules on which the structure of newly generate documents is based.

The internal structure identified so far is that **Pandoc** matches the length of equal signs and dashes that are used to *underline* headers to the character width of the headers, breaks normal text lines and fenced code regions become delimited with tabs.

### Standardising source documents

The existence of the internal structure in **Pandoc** means that any source document, that does not obey the internal structure, cannot be used in a invertible workflow. But documents that are already converted by **Pandoc**, i.e., *standardised*, will be consistent through stages of the workflow. Standardising of source documents is carried out by `standardise()`.

### Protecting R Code Chunks

Because **Pandoc** does not support R Markdown, R Code Chunks are not processed by **Pandoc** properly. As a result, R Code Chunks are changed with no apparent patterns. For example, the R Code Chunk in lines 6–8 of Listing 3.1 is changed like below:

```
‘{r, echo=FALSE} summary(cars)
```

The change does not provide a pattern on which reliable text processing can be based. So R Code Chunks are *protected* by enclosing them with special delimiters.

R Markdown format provides metadata sections in which title, author, date and options for customising output can be explicitly specified. This is a feature exclusive to R Markdown, which means metadata sections also require protections against **Pandoc**.

The function, `protect()`, protects R Code Chunks and metadata sections. To satisfy invertibility, standardised documents are *unprotected* by the function, `unprotect()`, that is, any text-based changes to protect are reverted.

## 3.3 Function details

### 3.3.1 `protect()`

An **R Markdown** document typically has a metadata section that contains information for title, author, date and options for customising output (such as theme and table of content availability). The metadata section is *exclusive* to the **R Markdown** format which means **Pandoc** does not recognise it as the regular Markdown syntax. As a result, this information is discarded after document conversion process by **Pandoc** takes place.

When **Pandoc** converts an **R Markdown** document into other document types, the R Code Chunks in the document are changed. R Code Chunks are specified by the native Markdown syntax for fenced code regions, thus one might expect them to be regarded as verbatim programming code throughout different document types. However R Code Chunks are modified with no clear patterns to the extent that no reliable text processing could be devised in order to fix the change.

Therefore metadata sections and R Code Chunks are protected, prior to the standardisation by **Pandoc**. Metadata sections are protected by enclosing them with `<!--rmd_metadata` and `rmd_metadata-->` `rmd-rmLines` delimiters, instead of their usual `---` delimiters. The delimiters used to wrap metadata sections are in the form of a comment in the HTML syntax as they are not interfered by **Pandoc** and perfectly retainable after standardisation. The text, `rmd-rmLines`, is used to account for the fact that **Pandoc** tends to remove empty lines between the end of R Code Chunks and text.

R Code Chunks are protected similarly by delimiting them with `<!--begin.keepcode` and `end.keepcode-->` markup. One difference here is that the delimiters are added in front and end of the usual ```` ```r ```` and ```` ``` ```` delimiters of R Code Chunks to wrap the entirety of R Code Chunks. This allows for more reliable

retention of the R code after standardising, as well as easier text processing.

The output of `protect()` is in `.pre.Rmd` format.

### 3.3.2 `standardise()`

**Pandoc** has an internally defined format that its output documents are based on. The internal format of **Pandoc** for a Markdown document requires that the length of text for header must be equivalent to the length of equal signs or dashes that *underline* the headers in setext-style, and code blocks must be delimited with tabs (or 4 spaces) instead of three backtick quotes.

When the inversion process by **Pandoc** is executed on a Markdown document that disobeys the internal format, the inverted document will always be different to the source document

As a result, the number of equal signs and dashes are shortened or extended to match the character width of the text for corresponding headers. Atx-style headers for first two header levels are converted to the setext-style header with the similar length matching, while the other four header levels remain the same.

Once a protected document is prepared, it is standardised into a format that remains consistent over repeated document conversion by **Pandoc**. By using the option, `--no-wrap`, **Pandoc** does not implement text-wrap, based on its internally defined rule.

### 3.3.3 `unprotect()`