Invertible Reproducible Documents

Eric Lim

October 9, 2014

Motivation

Knitr(REF) is a wonderful package that enables dynamic report generation with R. It allows integration 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).

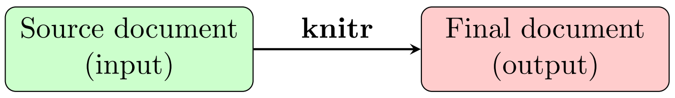


Figure 1.1: Unidirectional document generation by **knitr**

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 output), 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 *invertible*, as well as reproducible, that is, final reports must be able to be converted into the source document format without introducing any change from the inversion process itself.

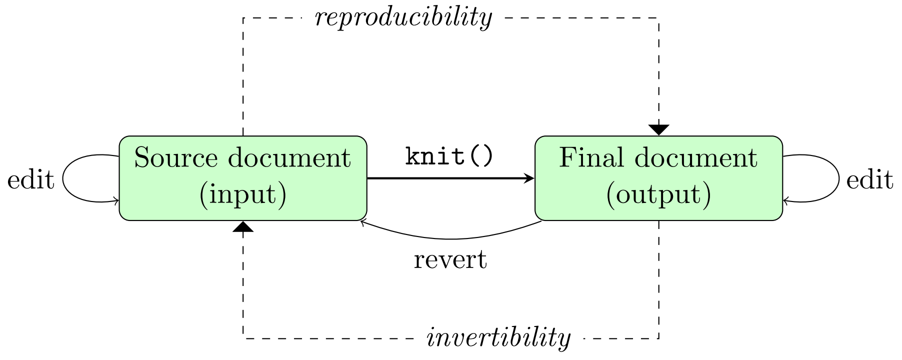


Figure 1.2: Invertible reproducible workflow

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

$$\begin{tikzcd}\boxed{\text{R HTML}} \arrow[r, "\texttt{knit()}"] & \boxed{\text{HTML}}\end{tikzcd}$$

2.1 Introduction

The primary objective of the initial phase is to design and achieve a complete reproducible, invertible workflow. There are many obstacles that limit the possibility of devising an invertible workflow. A sizable one of these problems is that converting (or inverting) between two different document formats requires a tremendous amount of a particular resource that we were limited on; *time*.

Out of the many formats that **knitr** supports, R HTML is the format we have chosen for the objective. There are two main reasons for this choice; R HTML does not require conversion between two document formats and it allows the use of powerful text processing tools, all of which increase our chance of accomplishing a reproducible, invertible workflow.

Support of unified format

In **knitr**, R HTML-based documents are *knitted* to produce HTML-based documents. The *only* difference between these two document formats is that R HTML-based documents contain sections of lines of R code, known as R Code Chunks. Here is an example of a simple R Code Chunk:

```
<!--begin.rcode
plot(cars)
end.rcode-->
```

By using the function, `knit()`, in **knitr**, the R Code Chunk is run in R to produce output (a plot for the previous example). The R code, `plot(cars)`, and the output plot are, then, enclosed separately in nested `div` elements in the final HTML-based document.

Knitr generates HTML documents, as output, from *HTML-based* R HTML documents, as input. In other words, the document generation process does not involve conversion between two distinct document formats. The source document format is essentially the same as the final document format.

By this property, difficulties associated with inverting a document format back into a different format can be minimised and the chance of inverting is increased.

Availability of tool sets

HTML is markup language very similar to XML. This means powerful XML-based tool sets, such as XPath (REF), are available for use to manipulate the HTML documents effectively. The R package, **XML** (REF), is used to provide these XML-based tool sets in R.

Demonstration

Here is a brief demonstration of how an R HTML document can be used to generate a HTML document.

Listing 2.1 is the code structure of the source document, `example1.Rhtml`. The highlighted text (lines 8–10) is an R Code Chunk. The option, `echo=FALSE`, is used to hide the R code in the final document.

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* the source document, `example1.Rhtml`, to generate the final document, `example1.html`:

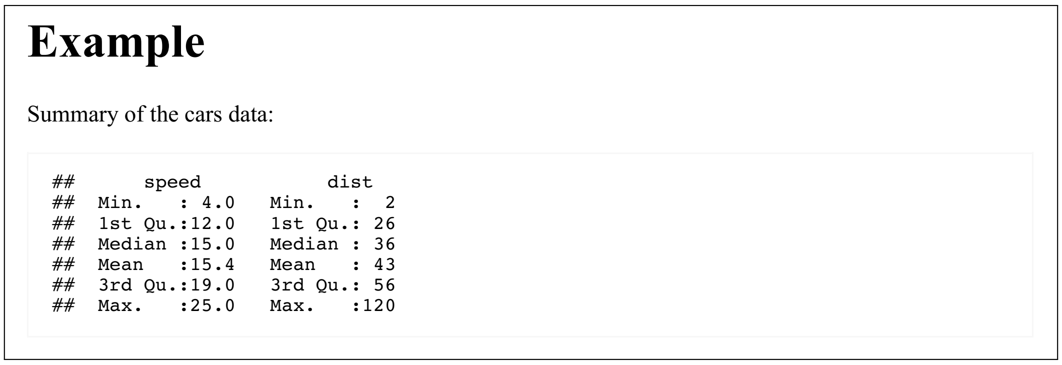
```
> library(knitr)
> knit("example1.Rhtml")
```

Listing 2.2 shows the code structure of the final document, `example1.html`. The R code from line 9 of Listing 2.1 is run in R to produce the result seen in lines 13–19 of Listing 2.2. Cautions must be taken as the code in Listing 2.2 has been tidied up with line endings to enhance readability. The actual untidied code consists of long lines that usually require text-wrap to fit into the page.

Listing 2.2: (tidied) `example1.html`

```
<!DOCTYPE html>
<html>
<head>
  <style type="text/css"> ... </style>
</head>
  <body>
    <h1>Example</h1>
    <p>Summary of the cars data:</p>
    <div class="chunk" id="unnamed-chunk-1">
      <div class="rcode">
      <div class="output">
        <pre class="knitr r">
          ##     speed          dist
          ##  Min.   : 4.0  Min.   :  2
          ##  1st Qu.:12.0  1st Qu.: 26
          ##  Median :15.0  Median : 36
          ##  Mean   :15.4  Mean   : 43
          ##  3rd Qu.:19.0  3rd Qu.: 56
          ##  Max.   :25.0  Max.   :120
        </pre></div>
      </div></div>
  </body>
</html>
```

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



Example

Summary of the cars data:

```
##      speed           dist
##  Min.   : 4.0   Min.   :  2
##  1st Qu.:12.0   1st Qu.: 26
##  Median :15.0   Median : 36
##  Mean   :15.4   Mean   : 43
##  3rd Qu.:19.0   3rd Qu.: 56
##  Max.   :25.0   Max.   :120
```

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.

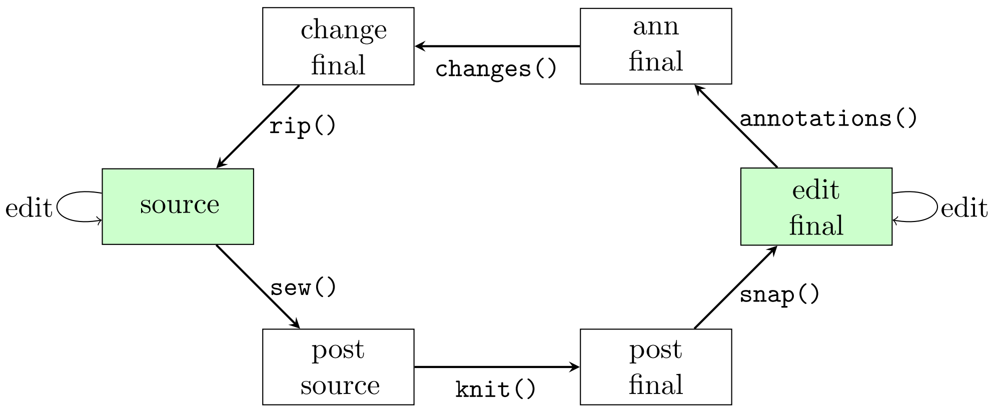


Figure 2.2: The reproducible, invertible workflow

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 a complete cycle.

The pre-processing task is carried out by the function, `sew()`.

Gain of information

As opposed to losing information, the document generation process can result in gain of new information. For example, lines 9–19 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 the post-processed (final) document. The sections of newly gained information, i.e., the output of R Code Chunks, may seem editable to the client.

However translating changes made on the gained information in 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-created sections of the final document to be annotated only. In this way, the client can still pass on feedbacks to the consultant without directly changing the R code or the output of the R code.

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 two libraries, **CKEditor** (REF) and **Annotator** (REF), that provide relatively user-friendly GUIs.

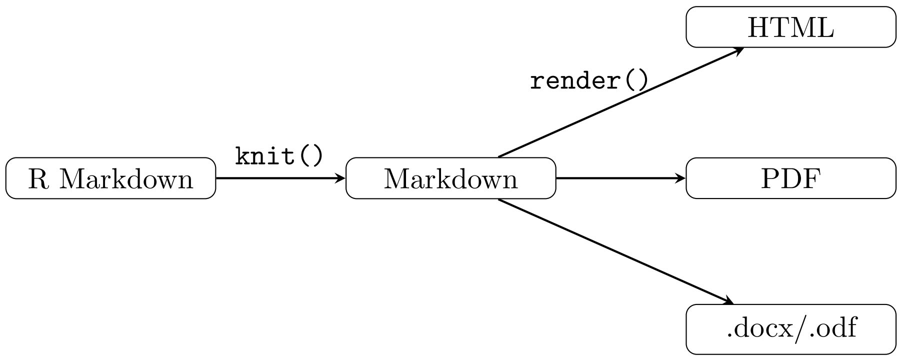
The JavaScript code is added by the function, `snap()`.

Merging changes

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

Phase 1 workflow is strictly limited to work on R HTML-based documents in conjunction with **knitr**. This limitation comes from the way our initial objective has been defined, that is, achieving the workflow (as described in Section 2.1). Because of this restriction, we do not know whether the workflow is *good* enough or viable elsewhere. It performs exactly as we have designed it to, but is it applicable to other formats?

An important aspect of the workflow to note is the *genericity.* A generic function means that it has specific methods for different types of objects, thus making it very robust. To write a generic piece of code, object-oriented programming is usually considered.

Object-oriented programming (REF) is one of the many programming paradigms that define the fundamental styles of computer programming. R language supports object-oriented programming very well through an *interactive* approach, as opposed to other programming languages, such as C and Java. In this style of programming, an *object* is created by a constructor function and is passed onto *methods* that carry out specific tasks for different types of objects. This means that an object-oriented software is more *generic* to handle different kinds of input more robustly than others.

Unfortunately, the functions invovled in the workflow are neither generic nor object-oriented. As a result, the functions can only handle R HTML- and HTML-based documents as input and phase 1 workflow is strictly limited to work on these two types of document formats. Generalising the workflow or making it object-oriented introduces two problems; the entire workflow may be subject to change, and there is uncertainty to determine if the workflow is worth going through the change.

Let us look at the problems in more detail. Since phase 1 workflow does not involve object-oriented programming, making it to be object-oriented means the entire workflow may require adjustments. Pursuing this will result in a completely different workflow that deviates from the original design we had in mind. In addition, there is no way to determine whether changing the workflow for the sole purpose of making it generic is truly beneficial. Before *any* attempts can be made to improve the workflow, it must be tested, at least, for its applicability. From this test, we can get a rough measure of its potential that may help decide whether to implement changes to the workflow.

In consideration of these problems, the next phase of the project has been decided to explore the applicability of the workflow. The format we have chosen for the second phase is R Markdown. There are four reasons behind our choice; flexibility of R Markdown, easy conversion to HTML, popularity and support.

Flexibility

R Markdown-based documents are *knitted* to be converted into Markdown (REF), which, in turn, can be converted into different document formats by document conversion software. Depending on the kind of the document converter, PDF, HTML and various other widely used document formats can be generated from Markdown. This flexibility provides the client with options to choose a document format that he or she feels comfortable using.

The document converter we have chosen is **Pandoc** (REF). **(Should I write the reasons for this choice?)**

Conversion to HTML

R Markdown-based documents can be naturally converted into HTML-based documents. This means that once a R Markdown-based source document is processed into a HTML-based final document, phase 1 workflow has a very high chance of being applicable without requiring much adjustment.

Popularity

Because R Markdown supports flexible conversion, it is increasingly popular and widely used. It has strong emphasis on readability from being an easy-to-read and -write plain text format. This simplicity further increases R Markdown's popularity to be a good candidate for our experiment.

Support for R Markdown

The latest version, R Markdown v2, has many improvements, one of which, for example, is the support for interactive document generation using **Shiny** (REF). This is a particularly useful feature in regards to the current global trend.

With the vast advancement in technology, the world demands evidence, often through means of visualising, before acceptance and acknowledgement. In the field of statistics, a particularly effective solution to this demand is via data visualisation, whose effectiveness can be exponentially enhanced by *interactivity*. An interactive visualisation method provides fun and exciting ways to visualise data through which client perception and learning can be greatly improved.

R Markdown's support for **Shiny**, as well as other improvements, is something that is very promising in securing its already popular usage. It also suggests that R Markdown is relatively active and responsive to meet global demands, thus giving us a reason to choose this document format for phase 2.

Demonstration

Listing 3.1 is the internal code structure of a simple R Markdown document, `example2.Rmd`. The highlighted text, lines 6–8, is a simple R Code Chunk.

Listing 3.1: `example2.Rmd`

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

There is a crucial difference between the two document generation processes of R HTML and R Markdown. R HTML-based documents are knitted *directly* into HTML, whereas R Markdown-based documents are knitted *intermediately* into Markdown, which is then *rendered* into a final format. Because there are *two* stages involved in one complete document generation process for R Markdown, the function `knit()` has to be used twice; firstly on the source R Markdown document and secondly on the knitted Markdown document. The difference is significant but can be overlooked. **Knitr** implements the use of the function, `render()`, in **rmarkdown** (REF) package which knits and renders in a single command.

The R code below calls `render()` to generate the HTML-based final document, `example2.html`:

```
rmarkdown::render("example2.Rmd", output_format="html_document")
```

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 15–21 of Listing 3.2. In general, the HTML document generated from R Markdown has more content, such as `meta` elements, than the one from R HTML. Note that there are more than one `meta`, `script` and `style` elements inside `head` but these are limited to one each for readability.

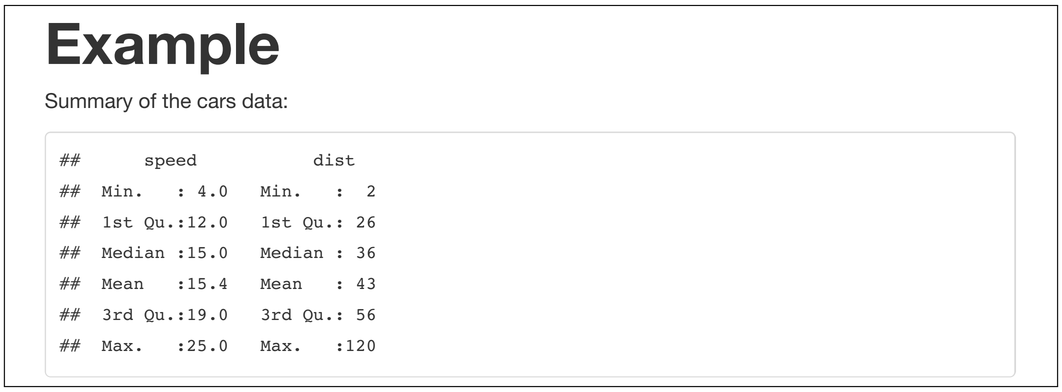
Even though the two final documents seen in Listings 2.2 and 3.2 are both in the HTML format, they are structurally quite different. Markup attributes such as `class="container-fluid main-container"` in line 10 of Listing 3.2 is different to `class="chunk"` markup in line 9 of Listing 2.2.

Listing 3.2: (tidied) `example2.html`

```
 1  <!DOCTYPE html>
 2  <html>
 3  <head>
 4    <meta ... />
 5    <script> ... </script>
 6    <style> ... </style>
 7  </head>
 8    <body>
 9      <style type="text/css"> ... </style>
10      <div class="container-fluid main-container">
11        <div id="example" class="section level1">
12          <h1>Example</h1>
13          <p>Summary of the cars data:</p>
14          <pre><code>
15            ##     speood         dist
16            ##  Min.   : 4.0  Min.   :  2
17            ##  1st Qu.:12.0  1st Qu.: 26
18            ##  Median :15.0  Median : 36
19            ##  Mean   :15.4  Mean   : 43
20            ##  3rd Qu.:19.0  3rd Qu.: 56
21            ##  Max.   :25.0  Max.   :120
```

```
        </code></pre>
      </div>
    </div>
    <script> ...bootstrap table styles... </script>
    <script> ... mathjax ... </script>
  </body>
</html>
```

Figure 3.1 shows what `example2.html` will look like in a web browser. Although nearly identical, we can see minor differences between the two final documents, seen in Figures 2.1 and 3.1, such as different font styles for the headings and spacing used for the R Code Chunk output. The differences are due to different `styles` used in both cases.



Example

Summary of the cars data:

```
##      speed           dist
##  Min.   : 4.0   Min.   :  2
##  1st Qu.:12.0   1st Qu.: 26
##  Median :15.0   Median : 36
##  Mean   :15.4   Mean   : 43
##  3rd Qu.:19.0   3rd Qu.: 56
##  Max.   :25.0   Max.   :120
```

Figure 3.1: `example2.html` in a browser

3.2 Overview

Three additional steps are introduced to phase 2 workflow as a pre-processing stage. Once the source document is *readied* by the three steps of pre-processing, it becomes reproducible and invertible.

Some of the function names kept identical to those in phase 1 workflow to minimise confusion and maintain simplicity. But it must be noted that there are extra features in these functions that work *exclusively* on R Markdown format.

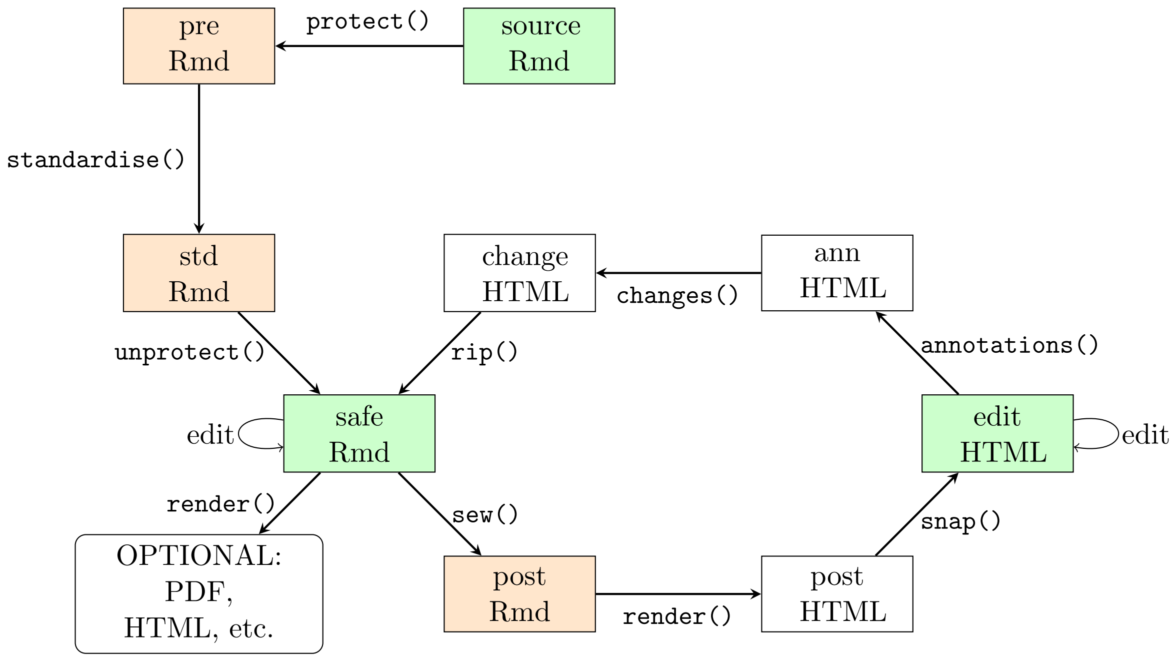
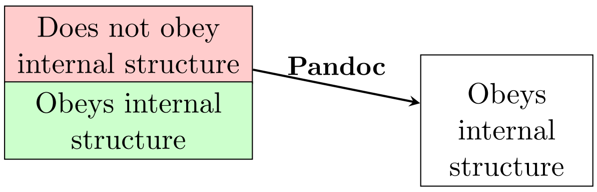


Figure 3.2: Revised reproducible, invertible workflow

Invertibility of Pandoc

Pandoc does not support flawless conversion between document formats, that is, the source document format is not the same as the final document format.

There is an internal structure in **Pandoc** that determines the structure of the output documents. This means that whether the source document obeys the internal structure or not, **Pandoc** will always produce a final document that obeys the internal structure. This is best understood with a simple diagram: (**Fix diagram**)



We have decided to use the internal structure as a means of *standardising* the source document so that any subsequent conversion by **Pandoc** does not interfere with its structure. The structural aspect of standardised documents will be consistent throughout the workflow.

The standardisation step is carried out by the function, `standardise()` with the use of **Pandoc**.

Loss of information

R Markdown has syntax for specifying metadata, such as author, title and date information. An example of a metadata section is as below:

```
---
title: "Example"
output:
  html_document:
    toc: true
    theme: united
---
```

Metadata sections are *exclusive* to R Markdown. As a result, **Pandoc** discards them in its output documents. This means that the metadata information cannot be carried over to subsequent steps after the standardisation step.

Conversion of formats