Invertible Reproducible Documents

Eric Lim

October 23, 2014

Motivation

Knitr (Xie et al., 2013) 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 (Knuth, 1984), 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. The importance of reproducibility can be further extended to be applied in the entire area of academia (Leeuw et al., 2001).

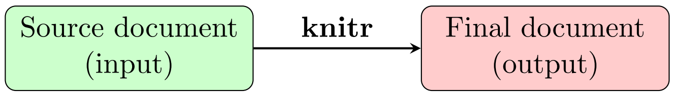


Figure 1.1: Unidirectional document generation by **knitr**

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.

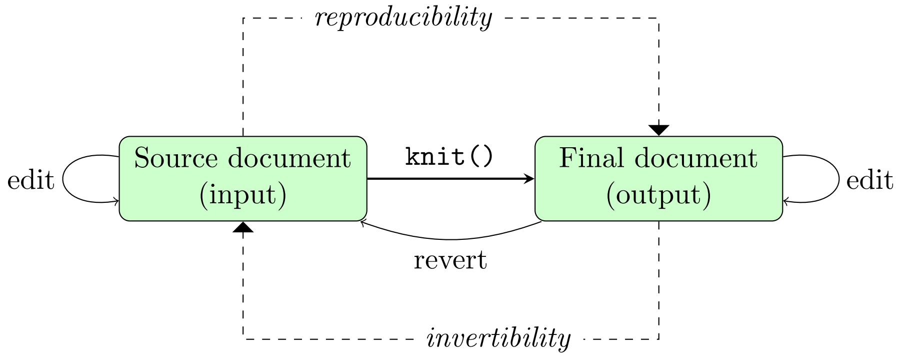


Figure 1.2: Invertible reproducible workflow

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

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

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 (Robie et al., 2014), are available for use to manipulate the HTML documents effectively. The R package, **XML** (Lang, 2013), 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`:

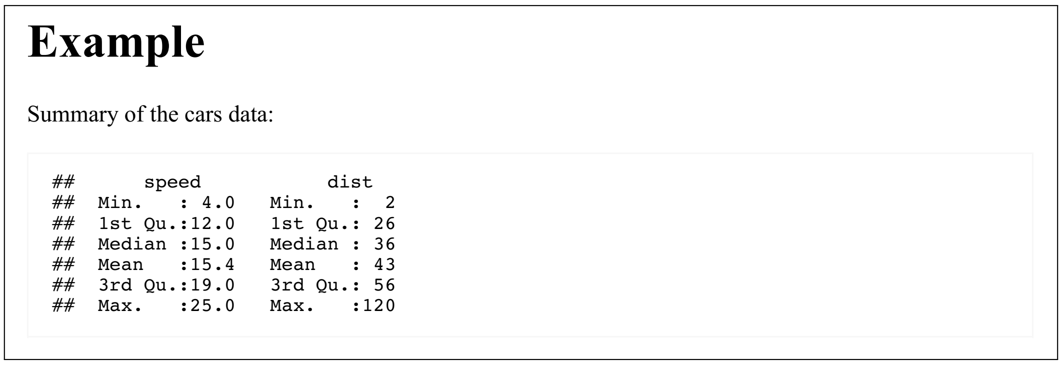
```
> library(knitr)
> knit("example1.Rhtml")
```

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`

```
 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">
10       <div class="rcode">
11       <div class="output">
12         <pre class="knitr r">
13           ##     speed          dist
14           ##  Min.   : 4.0  Min.   :  2
15           ##  1st Qu.:12.0  1st Qu.: 26
16           ##  Median :15.0  Median : 36
17           ##  Mean   :15.4  Mean   : 43
18           ##  3rd Qu.:19.0  3rd Qu.: 56
19           ##  Max.   :25.0  Max.   :120
20         </pre></div>
21       </div></div>
22   </body>
23 </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

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 actual code for the functions can be found in `https://github.com/elim1988/honours-project`.

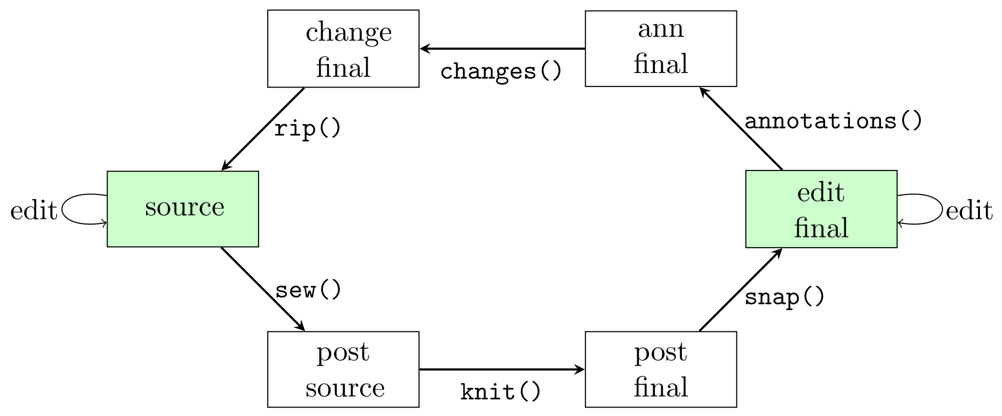


Figure 2.2: The reproducible, invertible workflow

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** (Knabben, 2014) and **Annotator** (Open Knowledge, 2014), 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()`.

Functions detail

`sew()`

The function carries out a pre-processing step, where R Code Chunks are copied. The copied R Code Chunks are delimited by `<!--begin.keepcode` and `end.keepcode-->` markup, and each copy is added *after* its original counterpart, correspondingly.

A simple example of a copied R Code Chunk is as below:

```
<!--begin.rcode echo=FALSE
summary(cars)
end.rcode-->
<!--begin.keepcode echo=FALSE
summary(cars)
end.keepcode-->
```

Inline R Code Chunks are delimited differently, by `<!--rinline.keep` and `-->` markup to differentiate them from the ordinary R Code Chunks.

The default suffix of the output file is `.post.Rhtml`, converted from the source suffix, `.Rhtml`.

`knit()`

The function is directly from the package, **knitr**, and carries out the usual knitting procedure. It should be noted that the copies of R Code Chunks generated from the previous step are *not* processed by `knit()`, and thus remain hidden in the subsequent steps.

The default suffix of the output file is `.html`, converted from the input suffix, `.Rhtml`.

`snap()`

The function is primarily involved with adding pieces of code to the (knitted) HTML document; it adds a piece of JavaScript code along with a piece of HTML script to the document and HTML attributes to the appropriate elements of the HTML document.

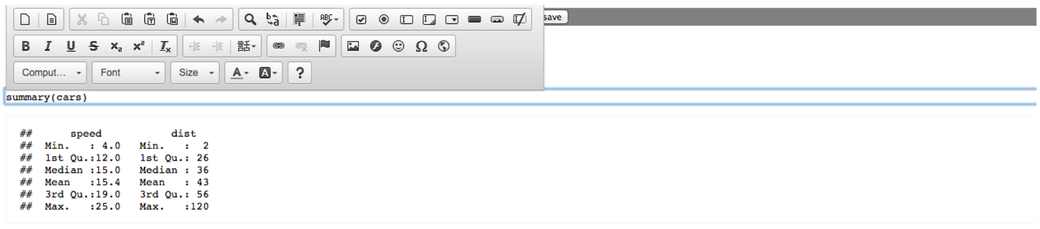
The JavaScript code contains `script` elements for loading the libraries, **CKEditor** and **Annotator**, responsible for providing the editable and annotatable GUIs, and **jQuery** (jQuery Foundation, 2014), on which the functionality of the former two libraries depends. The HTML script contains the layout of the save `button`.

The attributes, `contenteditable="true"` and `id="Editor"`, are added to each element in the document that should be editable, that is, any top level element that is *not* a `div` element with `class="chunk"` attribute. **CKEditor** detects elements with `contenteditable="true"` attribute and provides the editable GUI for *each* of them. The other attribute, `id="Editor"`, is used to allow reliable matching of the elements when changes are merged.

Correspondingly, **Annotator** detects `div` elements with `class="chunk"` attribute, which denote the sections of R-created content, and provides the annotatable GUI on *each* of these sections.

Once the final editable document is prepared using `sew()`, `knit()` and `snap()`, it must be hosted on a web server to save changes and annotations from **CKEditor** and **Annotator**. A test server has been set up at `http://stat220.stat.auckland.ac.nz/cke/` and can be accessed with the username and password, `"cke"`. Instructions can be followed on the web site to upload the editable final document prepared by `snap()`.

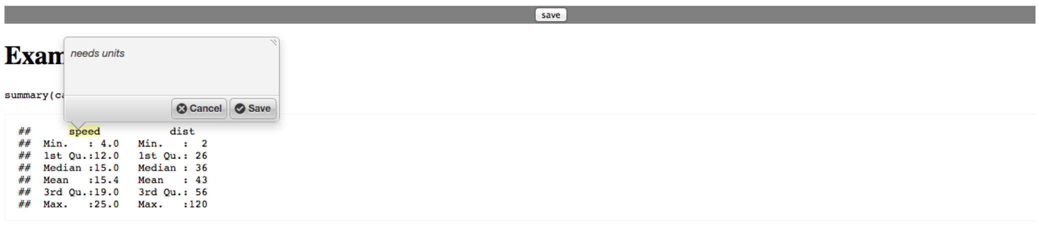
Figures 2.3 and 2.4 present the editable and annotatble GUIs. The line , *"Summary of the cars data:"* (Figure 2.1), is edited to *"summary(cars)"* in the Teletype font family. And the annotation, *"need units"*, is made on the text, *"speed"*, of the output. The "save" button on top of the page can be clicked to store the change and annotation separately in text files on the test server.



```
summary(cars)
```

```
##      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 2.3: CKEditor GUI



```
summary(c
```

```
##      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 2.4: `Annotator` GUI

The annotation is saved on a plain text file, `test-annotations.txt`, in **JSON** (Douglas Crockford, 2014) format and the change is saved on `test-`

`changes.txt` as simple text. The code structure of these two files can be seen in Listings 2.3 and 2.4.

Listing 2.3: `test-annotations.txt`

```
[
  {
    "ranges": [
    {
      "start":         "/div[1]/div[1]/pre[1]",
      "startOffset":    8,
      "end":           "/div[1]/div[1]/pre[1]",
      "endOffset":     13
      }
    ],
    "quote":"speed",
    "text":"needs units",
    "id":"61430634697899221413241555634"
  }
]
```

Listing 2.4: `test-changes.txt`

```
EDITOR Editor-1 NOT MODIFIED

EDITOR Editor-2:
<code>summary(cars)</code>
```

The output of `snap()` consists of one file, e.g., the editable final document with `.edit.html` suffix, converted from the input suffix, `post.html`. The editable document is, then, uploaded onto the test server where editing occurs.

When the editing is completed, cliking on the “save” button creates the two files, `test-annotations.txt` and `test-changes.txt`, on the test server for downloading and retrieval by other means.

`annotations()`

The function merges annotations from `test-annotations.txt` into the final document. The R package, **jsonlite** (Ooms et al., 2014), is the tool used to convert the JSON data (the annotations in `test-annotation.txt`) into an R object, so that they can be merged into the final document.

Since the R-created sections are restricted to be annotated only, i.e., clients are not allowed to edit them, they require manual adjustments. For this reason, annotations are merged as (HTML) paragraphs with highlighted text and are inserted at the top of their original locations (in the R-created sections), to ease the burden of looking for them and manual merging them afterwards.

The default suffix of the output file is `.anns.html`, converted from the input suffix, `.edit.html`.

Listing 2.5 is the code structure of the final document merged with the annotation. The highlighted text, lines 10–12, displays the merged annotation.

Listing 2.5: (tidied) `example1.anns.html`

```
 1  <!DOCTYPE html>
 2  <html>
 3  <head></head>
 4    <body>
 5      <p id="savebutton" style="background-color:grey; text-align:
          center">
 6        <button onclick="savechanges()">save</button>
 7      </p>
 8      <h1 contenteditable="true" id="Editor-1">Example</h1>
 9      <p contenteditable="true" id="Editor-2">Summary of the cars data
          :</p>
10      <p class="annotation" style = "background-color:coral">
11      <em>Annotation:  The text "speed" was annotated with the
12      message "needs units"</em></p>
13      <div class="chunk" id="unnamed-chunk-1">
14        <div class="rcode">
15        <div class="output">
16          <pre class="knitr r">
17            ##     speed          dist
18            ##  Min.   : 4.0  Min.   :  2
19            ##  1st Qu.:12.0  1st Qu.: 26
20            ##  Median :15.0  Median : 36
21            ##  Mean   :15.4  Mean   : 43
22            ##  3rd Qu.:19.0  3rd Qu.: 56
23            ##  Max.   :25.0  Max.   :120
24          </pre></div>
25        </div></div>
26    </body>
27  </html>
```

`changes()`

The function merges *changes* in the file, `test-changes.txt`, into the final document.

The *old* content, that has been edited, is replaced by the *new* content, in `test-changes.txt`.

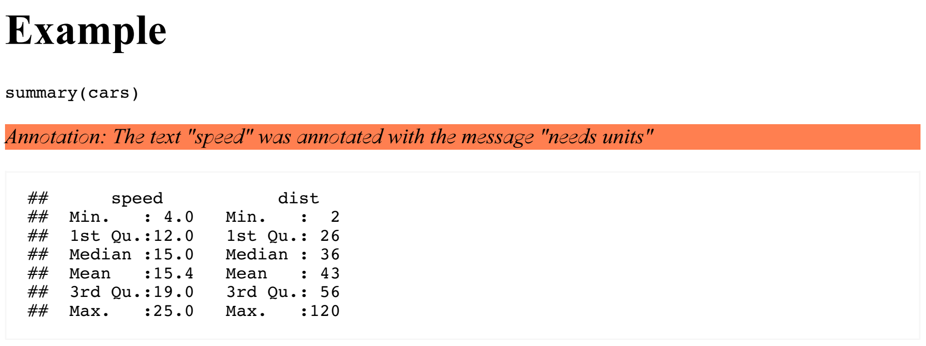
The default suffix of the output file is `.save.html`, converted from the input suffix, `.anns.html`.

Listing 2.6 is the code structure of the final document merged with the change. The highlighted text, line 10, shows the modified text.

Listing 2.6: (tidied) `example1.save.html`

```
 1  <!DOCTYPE html>
 2  <html>
 3  <head></head>
 4    <body>
 5      <p id="savebutton" style="background-color:grey; text-align:
            center">
 6        <button onclick="savechanges()">save</button>
 7      </p>
 8      <h1 contenteditable="true" id="Editor-1">Example</h1>
 9      <p contenteditable="true" id="Editor-2">
10  <code>summary(cars)</code>
11      </p>
12      <p class="annotation" style = "background-color:coral">
13      <em>Annotation: The text "speed" was annotated with the
14      message "needs units"</em></p>
15      <div class="chunk" id="unnamed-chunk-1">
16        <div class="rcode">
17        <div class="output">
18          <pre class="knitr r">
19            ##     speed          dist
20            ##  Min.   : 4.0   Min.   :  2
21            ##  1st Qu.:12.0   1st Qu.: 26
22            ##  Median :15.0   Median : 36
23            ##  Mean   :15.4   Mean   : 43
24            ##  3rd Qu.:19.0   3rd Qu.: 56
25            ##  Max.   :25.0   Max.   :120
26          </pre></div>
27        </div></div>
28    </body>
29  </html>
```

Figure 2.5 shows how the *annotated* and *changed* final document, `example1.save.html` looks like in a web browser.



Example

```
summary(cars)
```

Annotation: The text "speed" was annotated with the message "needs units"

```
##      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 2.5: `example1.save.html`

`rip()`

The function inverts final HTML-based documents, merged with changes and annotations, back into R HTML-based source documents.

More specifically, all unwanted contents added from the previous steps are removed. These include; the `script` and `style` elements added by `knit()` and `snap()`, the HTML attributes added by `snap()` and all R-created sections from `knit()`. The copies of R Code Chunks are reverted back to their original R HTML syntax, in place of the removed R-created sections, to become reproducible.

The default suffix of the output file is `.return.Rhtml`, converted from the input suffix, `.save.html`.

Listing 2.7 is the code structure of the inverted document, `example1.return.Rhtml`.

Listing 2.7: `example1.return.Rhtml`

```
<!DOCTYPE html>
<html>
<head>
</head>
  <body>
    <h1>Example</h1>
    <p>
<code>summary(cars)</code>
    </p>
    <p class="annotation" style = "background-color:coral">
```

```
11       <em>Annotation: The text "speed" was annotated with the message "needs units"</em>
12       <!--begin.rcode echo=FALSE
13       summary(cars)
14       end.rcode-->
15     </body>
16   </html>
```

Rather than manually comparing the original source and inverted documents, we can use `diff` utility in UNIX system. The command line below is to compare the original source file, `example1.Rhtml` and the inverted source file, `example1.return.Rhtml`:

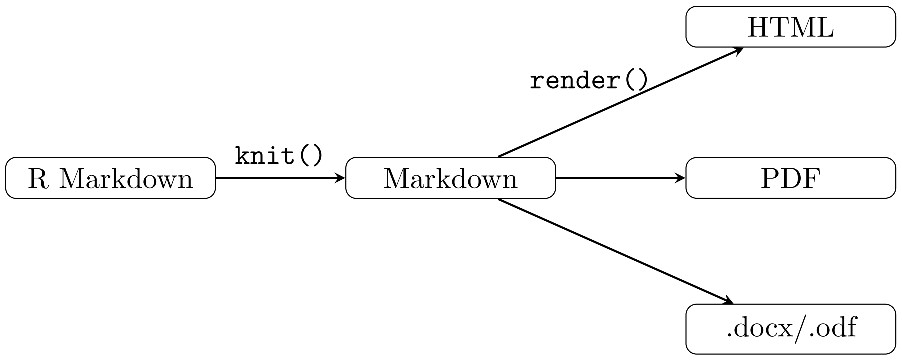
```
$ diff example1.Rhtml example1.return.Rhtml
```

Listing 2.8 is the output from the previous `diff` command. It says line 7 of the original source document (Listing 2.1) is changed with the lines 7–11 of the inverted file (Listing 2.7).

Listing 2.8: Output from `diff`

```
7c7,11
<     <p>Summary of the cars data:</p>
---
>     <p>
> <code>summary(cars)</code>
>     </p>
>     <p class="annotation" style = "background-color:coral">
>     <em>Annotation: The text "speed" was annotated with the message "needs units"</em>
```

Phase 2: R Markdown



3.1 Introduction

Phase 1 workflow has provided us with evidence to support the idea of implementing invertibility into reproducible document generation. However the workflow is strictly limited to work on R HTML-based documents, in conjunction with **knitr**.

R HTML has been the most suitable format to allow us to focus on exploring invertibility, which has been the primary objective of the first phase, but there are other document formats that are potentially more useful. It is of our strong interest to see if the workflow can be more generalised to be applicable for the other useful document formats.

The format we have chosen for the second phase is R Markdown (RStudio, 2014), which can be converted to Markdown (Gruber, 2014) using **knitr**. There are two main reasons behind our choice of the format; Markdown is rapidly becoming popular and many great tools are available for it.

Popularity

Markdown is a lightweight markup language that brings simple and readable formatting syntax, unlike the other more sophisticated languages such as TeX and HTML. Because of the simple and readable design, Markdown is gaining popularity amongst many who wish to avoid complexity.

The simplicity, however, does not affect the level of control on the Markdown-based documents as HTML-style markup can be incorporated into the documents to provide HTML exclusive features and controls.

Tools

Markdown's popularity has influenced many technologies to implement it in their frameworks. Some examples are PageDown (pag, 2014), which serves as a parser for Stack Overflow and Stack Exchange websites, and GitHub Flavored Markdown (GitHub, 2014), which is used across many features supported in the online project hosting website, GitHub. Many more implementations of Markdown exist across different platforms and languages.

A list of markdown implementations can be found at `http://www.w3.org/community/markdown/wiki/MarkdownImplementations`.

Many document conversion tools also exist for Markdown to offer flexible conversion into various document formats. What this flexibility allows us to implement in the workflow is the ability to produce final documents in different formats. For example, a final document can be converted to HTML and PDF, where the former can be published on the Web while the latter can be suitable for printing.

It is possible that our experiment of invertibility on Markdown could bring usable ideas to these implementations and vice versa, which is a sound reason to support our choice.

Demonstration

Listing 3.1 is the code structure of a simple R Markdown-based source document, `example2.Rmd`. The highlighted text, lines 13–15, is a R Code Chunk in the R Markdown syntax.

Listing 3.1: `example2.Rmd`

```
 1 ---
 2 title: ""
 3 output:
 4   html_document:
 5     toc: false
 6 ---
 7 
 8 Example
 9 =========================
10 
11 Summary of the cars data:
```

~~~~
12
13 ```{r}
14 summary(cars)
15 ```
~~~~

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 *converted* into the final format. Because there are *two* steps 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 as **knitr** implements use of the function, `render()`, from **rmarkdown** (RStudio, 2014) package which generates final documents 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, `example2.html`. The original R Code Chunk in lines 13–15 of Listing 3.1 is processed to produce the result seen in lines 13–19 of Listing 3.2. A notable difference is that all elements are now nested in a top-level `div` element with the class attribute of `"container-fluid main-container"`.

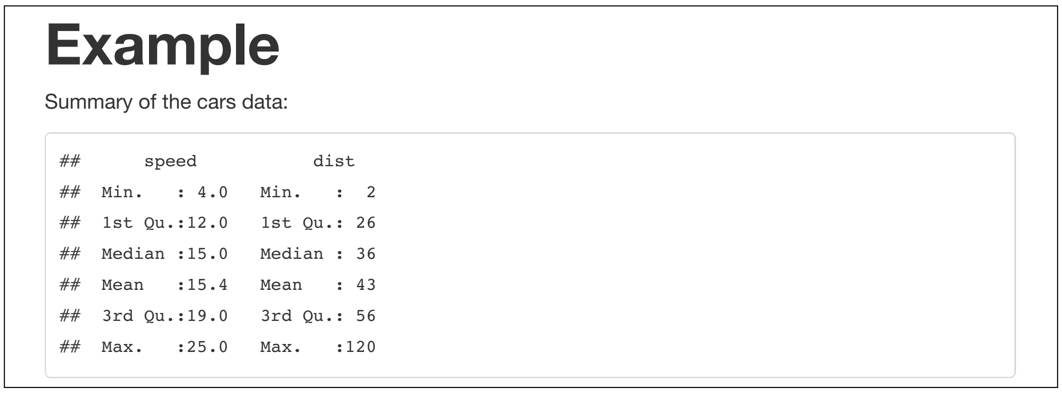
In general, the HTML document generated from R Markdown has more content, such as `meta` elements, than the one from R HTML. Note that there are more than one `meta`, `script` and `style` elements inside the `head` element but these are kept brief in line 4 for simplicity.

Listing 3.2: (tidied) `example2.html`

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

```
        ##     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
      </code></pre>
    </div>
  </div>
  <script> ...Bootstrap table styles... </script>
  <script> ...  MathJax ... </script>
 </body>
</html>
```

Figure 3.1 shows how `example2.html` looks 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. These minor differences are due to different `style` and `script` elements 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 a source document is pre-processed by the three steps, it becomes reproducible and invertible, and is ready to enter the workflow.

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

The actual code for the functions can be found at `https://github.com/elim1988/honours-project`...

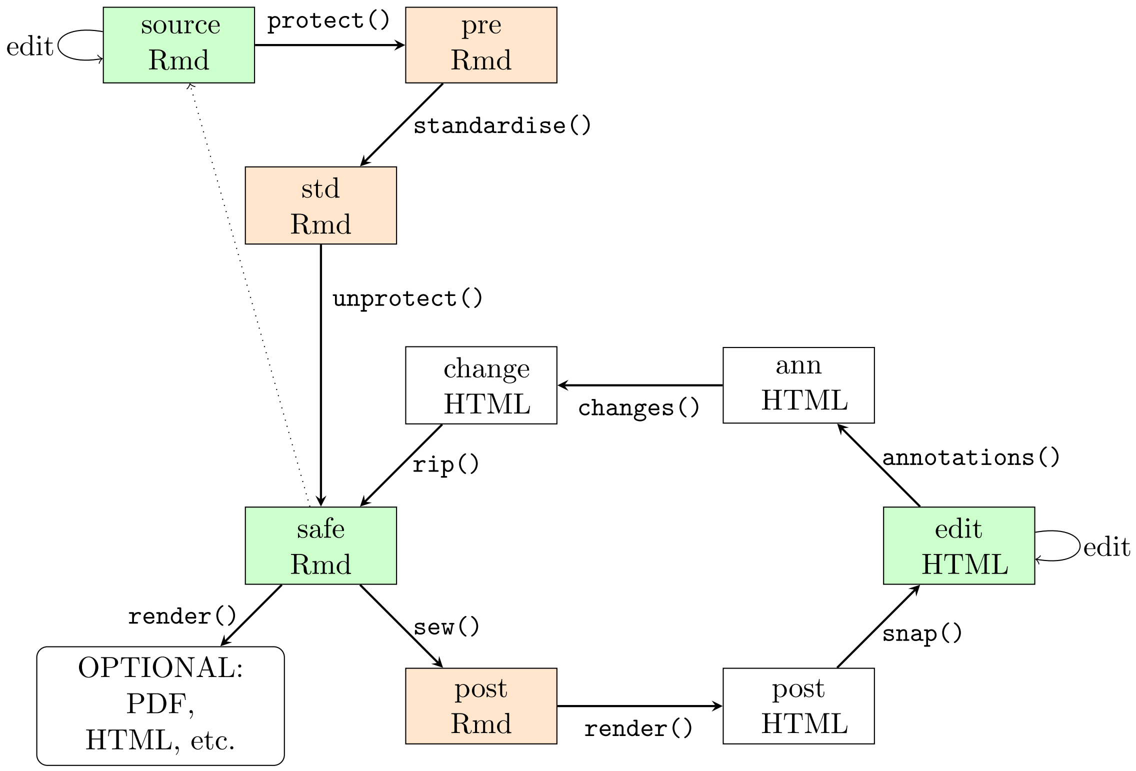


Figure 3.2: Revised reproducible, invertible workflow

Structural differences

Two final documents of the same format, each generated from `knit()` and `render()`, are structurally different. The differences exist for the case of HTML, because R Markdown uses tools such as **MathJax** (mat, 2014), which allows mathematical equations to be specified in LaTeX-style syntax rather than the usual HTML maths syntax, and **Bootstrap** (boo, 2014), which allows more precise customisation of the layout of the HTML document.

As a result, the *rendered* final document contains more `script` and `style` elements than its *knitted* counterpart, as well as different tree structures.

The solution we have chosen to compensate for the differences is to introduce additional steps in the workflow and implement different rules, such as differnt XPath, for the functions of phase 2. These will be discussed in more detail.

Conversion of formats

Pandoc (Macfarlane, 2014) is the document conversion tool chosen to be implemented in phase 2 workflow due to one main reason; it is fully supported by **knitr**. In fact, the latest version of **knitr** comes *with* **Pandoc**.

It can provide conversion of HTML into Markdown, which is necessary for the final step of the workflow involving the function, `rip()`.

Invertibility of Pandoc

Pandoc does not support flawless round trip between two document formats, that is, a source document of one format is not the same as its inverted counterpart of the same format. Figure 3.3 provides a simple diagram to visualise this using Markdown and HTML as the source and intermediary formats, respectively.

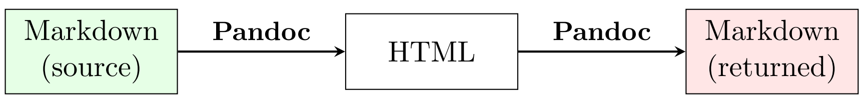


Figure 3.3: The two Markdown documents are structurally different.

The cause of this occurrence is due to the existence of an internal structure in **Pandoc** that determines the structure of its output documents. This means that whether the source document obeys the internal structure or not, **Pandoc** will always produce a final document that follows the internal structure, i.e., it *standardises*. This is best understood with a simple diagram:

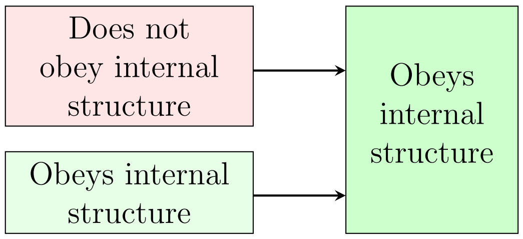


Figure 3.4: Standardisation of **Pandoc**

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 integrity of standardised documents will be consistent throughout the workflow.

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

Editing source documents

Because of the requirement for standardisation, an inverted document must be standardised each time it runs through the workflow. This is indicated by the dotted arrow in Figure 3.2.

Loss of information 2

R Markdown has syntax for specifying (YAML) 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. **Pandoc** also interferes with R Code Chunks in R Markdown, because R Code Chunks are not known to **Pandoc**. The solution we have chosen is to implement a pre-processing step to protect the metadata sections and R Code Chunks.

The usual delimiters, `---`, for metadata are changed to `<!--rmd_metadata` and `rmd_metadata--> rmd-rmLines`. The protected version of the previous metadata example is as below:

```
<!--rmd_metadata
title: "Example"
output:
  html_document:
    toc: true
    theme: united
rmd_metadata--> rmd-rmLines
```

R Code Chunks are protected by `<!--begin.keepcode` and `end.keepcode--> rmd-rmLines` delimiters.

The function, `protect()`, carries out the protection of metadata and R Code Chunks.

Optional pathway

Once the final version of a document is achieved, there is an optional path using the function, `render()`, to produce the final document in different formats.

The optional path is indicated by the “OPTIONAL” node in Figure 3.2.

3.3 Functions detail

`protect()`

The function *protects* metadata sections and R Code Chunks in the source document.

Metadata sections are delimited by `<!--rmd_metadata` and `rmd_metadata--> rmd-rmLines`.

R Code Chunks are delimited by `<!--begin.keepcode` and `end.keepcode--> rmd-rmLines`.

The use of `rmd-rmLines` is to correct for **Pandoc**'s tendency of removing empty lines after the protected lines.

The default suffix of the output file is `.pre.Rmd`, converted from the source suffix, `.Rmd`.

Listing 3.3 is the code structure of the protected source document, `example2.pre.Rmd`.

Listing 3.3: `example2.pre.Rmd`

~~~
<!--rmd_metadata
title: ""
output:
  html_document:
    toc: false
rmd_metadata--> rmd-rmLines

Example
=========================

Summary of the cars data:

<!--begin.keepcode```{r, echo=FALSE}
summary(cars)
```end.keepcode--> rmd-rmLines
~~~

`standardise()`

The function calls **Pandoc** to *standardise* the protected source document. In this function, `system()` is used to invoke the command line necessary for calling **Pandoc**.

The default suffix of the output file is `.std.Rmd`, converted from the input suffix, `.pre.Rmd`.

Listing 3.4 is the code structure of the standardised source document, `example2.std.Rmd`. A couple of things to note is that the equal signs in line 10, which correspond to R Markdown's syntax for specifying a heading, are reduced in length, and the delimiters, `rmd-rmLines`, are broken into the next lines in lines 7 and 17. These are some of the examples of **Pandoc**'s internal structure.

Listing 3.4: `example2.std.Rmd`

```
 1  <!--rmd_metadata
 2  title: ""
 3  output:
 4    html_document:
 5      toc: false
 6  rmd_metadata-->
 7  rmd-rmLines
 8
 9  Example
10  =======
11
12  Summary of the cars data:
13
14  <!--begin.keepcode```{r, echo=FALSE}
15  summary(cars)
16  ```end.keepcode-->
17  rmd-rmLines
```

`unprotect()`

The function removes the protection added by `protect()`.

The special delimiters used to protect metadata and R Code Chunks are reverted back to their normal R Markdown syntax.

The pushed down lines for `rmd-rmLines` delimiters are simply removed.

The default suffix of the output file is `.safe.Rmd`, converted from the input suffix, `.std.Rmd`.

Listing 3.5 is the code structure of the standardised invertible document, `example2.safe.Rmd`.

Listing 3.5: `example2.safe.Rmd`

```
---
title: ""
output:
  html_document:
```

```
 5     toc: false
 6 ---
 7 
 8 Example
 9 =======
10 
11 Summary of the cars data:
12 
13 ‘‘‘{r, echo=FALSE}
14 summary(cars)
15 ‘‘‘
```

We can use the `diff` command below to compare the source document, `example2.Rmd`, and the invertible source document, `example2.safe.Rmd`:

```
$ diff example2.Rmd example2.safe.Rmd
```

Listing 3.6 is the output form the previous `diff` command. The only difference between the two source documents is line 9, where the number of equal signs is reduced by the internal structure of **Pandoc**.

Listing 3.6: Output from `diff`

```
9c9
< =========================
---
> =======
```

`sew()`

The function carries out a similar pre-processing step as the phase 1 version with one addition; it copies metadata sections (as well as R Code Chunks) and append the copies after their originals.

The default suffix of the output file is `.post.Rmd`, converted from the input suffix, `.safe.Rmd`.

`render()`

The function is directly from the R package, `rmarkdown`. This function is what **knitr** uses to convert R Markdown-based source documents into final documents. In fact, the latest version of **knitr** is included with **rmarkdown**.

The default suffix of the output file is `.post.html`, converted from the input suffix, `post.Rmd`.

snap()

The function carries out conceptually the same tasks as the phase 1 version; add pieces of HTML and JavaScript code to the final document, and add HTML attributes to editable nodes. But different rules are used for phase 2 snap().

The JavaScript code, that loads **jQuery**, **CKEditor** and **Annotator**, is appended at the end of other `script` elements that are added by `render()` to avoid interference from them.

Because of the differences in the tree structure, the location of editable elements is different. They are nested in a top-level `div` element with the `class` attribute, `"container-fluid main-container"`. If there is at least one heading, all elements under the heading are further nested inside another `div` element with `id` attribute of the heading name. Thus different XPath is used to locate the editable elements and add `contenteditable` and `id` attributes accordingly.

Once the editable document is successfully obtained, it can be uploaded on the test server for editing, similarly to the phase 1 procedure.

The default suffix of the output file is `.edit.html`, converted from the input suffix, `.post.html`.

Listing 3.7 is the code structure of the editable final document, `example2.edit.html`.

Listing 3.7: (tidied) example2.edit.html

```
<!DOCTYPE html>
<html>
<head>
  various meta, script and style elements.
</head>
  <body>
    <style type="text/css"> ... </style>
    <div class="container-fluid main-container">
      <div id="example" class="section level1">
        <!--rmd_metadata
        title: ""
        output:
          html_document:
            toc: false
        rmd_metadata-->
        <h1 contenteditable="true" id="Editor-1">Example</h1>
        <p contenteditable="true" id="Editor-2">Summary of the cars
           data:</p>
```

~~~
        <pre><code>
          ##     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
          <!--begin.keepcode```{r, echo=FALSE}
          summary(cars)
          ```end.keepcode-->
        </code></pre>
      </div>
    </div>
    <script> ...Bootstrap table styles... </script>
    <script> ...  MathJax ... </script>
  </body>
</html>
~~~

`annotations()`

Only one aspect of the function differs from its phase 1 counterpart due to the differences in the tree structure.

All R-created contents are included in `code` elements, nested in `pre` elements, instead of `div` elements with `class="chunk"` attribute for phase 1. Hence different XPath is used to locate these sections of R-created contents and insert annotations accordingly.

The default suffix of the output file is `.anns.html`, converted from the input suffix, `.edit.html`.

Listing 3.8 is the code structure of the annotated final document, `example2.anns.html`, using the same annotation as the previous example.

Listing 3.8: (tidied) `example2.anns.html`

```
<!DOCTYPE html>
<html>
<head>
  various meta, script and style elements.
</head>
  <body>
    <p id="savebutton" style="background-color:grey; text-align:
        center">
      <button onclick="savechanges()">save</button>
```

~~~
  </p>
  <style type="text/css"> ... </style>
  <div class="container-fluid main-container">
    <div id="example" class="section level1">
      <!--rmd_metadata
      title: ""
      output:
        html_document:
          toc: false
      rmd_metadata-->
      <h1 contenteditable="true" id="Editor-1">Example</h1>
      <p contenteditable="true" id="Editor-2">Summary of the cars
          data:</p>
          <p class="annotation" style = "background-color:coral">
          <em>Annotation:  The text "speed" was annotated with the
          message "need units"</em></p>
      <pre><code>
        ##     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
        <!--begin.keepcode```{r, echo=FALSE}
        summary(cars)
        ```end.keepcode-->
      </code></pre>
    </div>
  </div>
  <script> ...Bootstrap table styles... </script>
  <script> ...  MathJax ... </script>
 </body>
</html>
~~~

`changes()`

The function is identical to its phase 1 counterpart, except that it uses different XPath to correct for the different tree structure.

The default suffix of the output file is `.save.html`, converted from the input suffix, `.anns.html`.

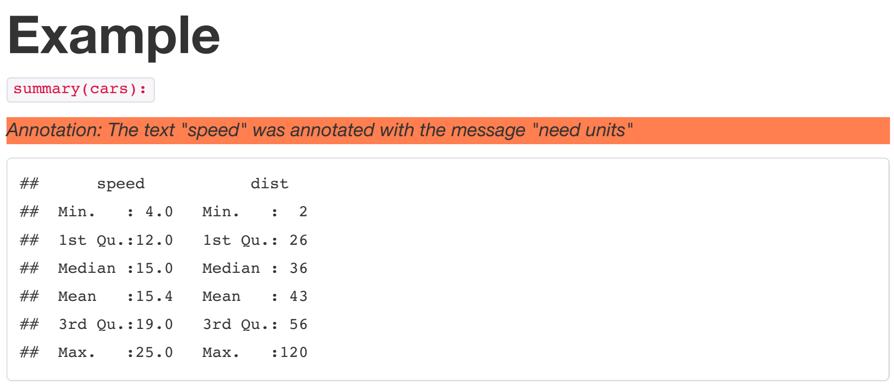
Listing 3.9 is the code structure of the final document merged with the change, same as the previous example.

Listing 3.9: (tidied) `example2.save.html`

~~~
<!DOCTYPE html>
<html>
<head>
  various meta, script and style elements.
</head>
  <body>
    <p id="savebutton" style="background-color:grey; text-align:
        center">
      <button onclick="savechanges()">save</button>
    </p>
    <style type="text/css"> ... </style>
    <div class="container-fluid main-container">
      <div id="example" class="section level1">
        <!--rmd_metadata
        title: ""
        output:
          html_document:
            toc: false
        rmd_metadata-->
        <h1 contenteditable="true" id="Editor-1">Example</h1>
        <p contenteditable="true" id="Editor-2">
        <code>summary(cars):</code>
        </p>
            <p class="annotation" style = "background-color:coral">
            <em>Annotation: The text "speed" was annotated with the
            message "need units"</em></p>
        <pre><code>
          ##     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
          <!--begin.keepcode```{r, echo=FALSE}
          summary(cars)
          ```end.keepcode-->
        </code></pre>
      </div>
    </div>
~~~

```
    <script> ...Bootstrap table styles... </script>
    <script> ...  MathJax ... </script>
  </body>
</html>
```

Figure 3.5 shows how the *annotated* and *changed* final document, `exampe2.save.html` looks like in a web browser.


Example

`summary(cars):`

Annotation: The text "speed" was annotated with the message "need units"

```
##      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 3.5: `example2.save.html`

`rip()`

The function carries out *two* additional steps; one of which uses **Pandoc** to remove the unwanted contents and the other to tidy up before inverting source documents.

The first stage of the function is to remove *some* of the unwanted contents, such as the two attributes added to editable elements.

The second stage uses **Pandoc** to remove the rest of the unwanted contents, in a similar manner to the standardisation step of the workflow. This method has been implemented in `rip()` because **Pandoc** removes the unwanted contents, such as various `script` and `style` elements relatively easily and much more robustly than the text processing approach used in phase 1.

The third stage involves adjustments necessary to *tidy* up the document to complete the inversion process.

The default suffix of the output file is `.return.Rmd`, converted from the input suffix, `.save.html`.

Listing 3.10 is the code structure of the inverted document, `example2.return.Rmd`. Note that the back-ticks seen in line 11 are the syntax for specifying code style, and the asterisks seen in line 13 are the syntax for italicising.

Listing 3.10: `example2.return.html`

```
---
title: ""
output:
  html_document:
    toc: false
---

Example
=======

‘summary(cars):‘

*Annotation: The text "speed" was annotated with the message "need
units"*

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

And finally, the `diff` command below can be used to compare the source and the inverted documents, `example2.Rmd` and `example2.return.Rmd`.

```
$ diff example2.Rmd example2.return.Rmd
```

Listing 3.11 is the output from the previous `diff` command that suggests the only differences are the lines corresponding to the merged change and annnotation.

Listing 3.11: Output from `diff`

```
11c11,13
< Summary of the cars data:
---
> ‘summary(cars):‘
>
> *Annotation: The text "speed" was annotated with the message "
need units"*
```

Discussion

We have described a couple of workflows, each of which is specific to work with one particular type of source document, that allows a consultant to write a report as invertible and reproducible, a client to edit and annotate directly on the report and the corrected report to be inverted back to the source format with the change and annotation merged.

Phase 1 workflow has brought some plausible ideas that an invertible workflow can be implemented in reproducible document generation. By no means is the workflow production-quality nor can it be readily substituted for another pre-exisiting well established workflow, but its strength lies in demonstrating the concept of invertibility and providing a departing point for further exploration.

Phase 2 workflow has provided some evidence towards the possibility of generalising the invertible workflow. Again it is not of production-quality, but it has shown that further generalisation is achievable with greater effort.

The rest of this chapter includes limitations associated with the workflows, comparisons with other workflows and tools, and suggestions for generalising them.

Limitations

The main limitation comes from the fact that the functions, involved in the workflows, are tightly *coupled* with the tools implemented in the workflows, i.e., their functionality depends on the use of this particular combination of the tools.

A direct consequence from the high coupling is that the functions are not very general; they are *specific* to work only with **knitr**, **CKEditor**, **Annotator** and **Pandoc**. An attempt at replacing these tools with different tools will not be successful unless appropriate adjustments are made for the functions.

Another consequence is that the functions are fragile; they are defenceless against changes in the underlying tools. Most of the functions work on the assumptions, or expectations, that these tools behave in certain *patterns*, e.g., **Annotator** only works on particular `div` elements that should be annotatable based on the assumption that `knit()` consistently puts all R-created contents in these elements. It can be problematic if changes are introduced in the tools, possibly with an update, to alter these assumptions.

Other limitations include difficulties with providing editable environment for images, tables and equations and dealing with these elements in general. These particular limitations could be associated with our choice of HTML as the format, as opposed to TeX-based Rnw format that may easily avoid these limitations. However HTML offers benefits such as *interactivity* which is not as easily accessible using Rnw.

Phase 2 specific limitations are that the functions are not based on all of the R Markdown's syntax, such as HTML markup within R Markdown, and that only some of the internal structure of **Pandoc** has been considered.

All of these limitations could be potentially avoided by generalising the workflows so that alternative more feature-rich tools can be implemented. A possible solution to improve the fragile defence of the functions is to implement an automated syncing process in them so they can always work on most up-to-date patterns.

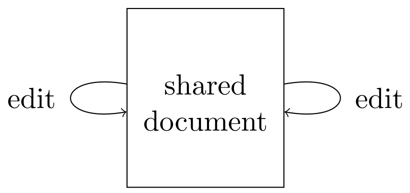
Other workflows and tools

Phase 1 and 2 workflows are designed to implement the idea of invertibility; their design focus is predominantly in inverting final documents.

Beyond this restrictive design, there are many reproducible document workflows, which may not be invertible but are worth introducing for comparisons and to generate ideas for future direction of the project.

Also there are many alternative tools to **knitr** for consideration.

Shared document workflow



Shared document workflows involve editing on a single document, i.e., the source and final documents are the same. Once a final document is generated, it can be edited indefinitely. A notable package in R to implement a shared document workflow is **SWord** (REF), where final documents are generated as Word documents. Another tool is **RExcel** (REF) that provides a shared document workflow in Excel.

This type of workflow is best suited for collaborative work that requires no version control, e.g., collaborative research amongst professionals where changes do not require thorough examination and investigation before merging.

The fundamental problem that we are trying to solve in this project, that is the aforementioned client-consultant situation, is completely avoided in a shared document workflow; clients and consultants work on the same document. The whole purpose of being a client or a consultant is defeated.

However the shared document workflow is usually more efficient as it does not require inversion steps, and collaboration makes them more productive. These advantages are something to consider for the invertible workflows to implement in future.

Beyond the R environment, **IPython notebooks** (development team, 2014) and **R Extensions for MediaWiki** (rex, 2014) are two notable tools that implement shared document workflows.

Workflows based on other formats

There are countless packages and tools to provide reproducible document workflows based on different document formats.

One example is an R package, **odfWeave** (from Steve Weston et al., 2014), which provides an Open Office-based reproducible workflow.

More R packages to provide alternative workflows can be found at `http://cran.r-project.org/web/views/ReproducibleResearch.html`.

Predecessors of knitr

The dynamic report generation tool, **knitr**, is essentially the summation of various pre-existing packages along with improvements and add-ons. These pre-dating packages include **Sweave** (Leisch, 2002) and **R2HTML** (Lecoutre, 2003), both of which offer reproducible document generation based on HTML as final documents.

Depending on the structural complexity of the final documents generated using these tools, it seems highly plausible that the invertible workflows can work with either or both of these tools with minor adjustments.

Alternatives of knitr

R packages such as **hwriterPlus** (Scott, 2011) and **ReporteRs** (Gohel, 2014) also offer reproducible report generation in HTML.

ReporteRs requires particular attention as it can add R plots as editable vector graphics in the final documents to allow viewers to directly modify the plots.

The difficulties associated with images and plots could be overcome by implementing such a package in an invertible workflow.

Generalisation

Generalising the invertible workflows can help relieve their restrictions to work only with HTML documents, which can potentially improve their usability.

This section considers possibility of generalising the invertible workflows.

Beyond CKEditor and Annotator

It may be obvious that many other JavaScript libraries are better and more robust than **CKEditor** and **Annotator**. But there are useful aspects of these libraries which might not be present in others.

A useful aspect of **CKEditor** is the ability to provide an editable GUI for *each* section, which might be a difficult feature to replicate in other tools.

Annotator has been useful as it provides more detailed information than needed, which is more advantageous than lacking in information.

It may be worthwhile to consider whehter JavaScript libraries to replace **CKEditor** and **Annotator** can offer similar features beforehand.

Beyond HTML

Although phase 2 workflow has suggested that Markdown-based workflows can be invertible, there are still many other document formats to consider.

One of the most desirable format is TeX-based Rnw. However it seems highly unlikely that Rnw-based invertible workflow is possible due to complications in inverting PDF files into Rnw-based source code.

Beyond client-consultant situation

The invertible workflows described in this report are motivated by the previously mentioned client-consultant scenario.

An invertible workflow is suitable for this particular situation because the levels of programmatic and technical skills vary widely between clients and consultants.

They can be extended to bring relevance in other contexts such as scientific reproducibility, where similar situations can often arise.

Extending R

Having to annotate on the sections of R-created contents is a restriction that is only a temporary solution to deal with the issues of losing and gaining of information.

If R itself can be extended to be hosted on the test server, in a similar manner to **R Extensions for MediaWiki**, R Code Chunks can be directly edited and run to observe different output.

Extremely robust version control system must then be implemented as it can be difficult to keep track of which code is changed or left alone.

Conclusion

Our ultimate goal remains to be in further generalising the round trip to suit a broader spectrum of document types.

Phase 1 workflow has contributed evidence that invertible reproducible document generation is possible.

Phase 2 workflow has suggested that the invertible reproducible document generation can be general to a certain degree.

Although these workflows are not to be readily integrated into commercial world, the ideas described in this report are hopefully useful for people who are passionate about document generation and value reproducible research.

Contributions

The project supervisors contributed with various ideas and guidance.

The functions, `sew()`, `snap()`, `annotate()`, `changes()` and `rip()`, are developed by the author.

Paul Murrell wrote the JavaScript code for loading **CKEditor** and **Annotator**, and set up the test server.

Finlay Thompson provided the original problem statement.

Further reading

Bibliography

Bootstrap JavaScript library, 2014. URL http://getbootstrap.com/.

MathJax JavaScript library, 2014. URL http://www.mathjax.org/.

PageDown, 2014. URL https://code.google.com/p/pagedown/wiki/PageDown.

R Extention for MediaWiki, 2014. URL http://mars.wiwi.hu-berlin.de/mediawiki/sk/index.php/R_Extension_for_MediaWiki.

IPython development team. IPython, 2014. URL http://ipython.org/notebook.html.

Douglas Crockford. JSON, 2014. URL http://json.org/.

Max Kuhn. Contributions from Steve Weston, Nathan Coulter, Patrick Lenon, Zekai Otles, and the R Core Team. *odfWeave: Sweave processing of Open Document Format (ODF) files*, 2014. URL http://CRAN.R-project.org/package=odfWeave. R package version 0.8.4.

GitHub. GitHub Flavored Markdown, 2014. URL https://help.github.com/articles/github-flavored-markdown/.

David Gohel. *ReporteRs: Microsoft Word, Microsoft Powerpoint and HTML documents generation from R*, 2014. URL http://CRAN.R-project.org/package=ReporteRs. R package version 0.6.0.

John Gruber. Markdown, 2014. URL http://daringfireball.net/projects/markdown/.

jQuery Foundation. jquery javascript library, 2014. URL http://jquery.com/.

Frederico Knabben. CKEditor JavaScript library, 2014. URL `http://www.w3.org/TR/xpath-30/`.

Donald E. Knuth. Literate programming. *The Computer Journal*, 27:97–111, 1984.

Duncan Temple Lang. *XML: Tools for parsing and generating XML within R and S-Plus.*, 2013. URL `http://CRAN.R-project.org/package=XML`. R package version 3.98-1.1.

Eric Lecoutre. The R2HTML package. *R News*, 3(3):33–36, December 2003. URL `http://cran.r-project.org/doc/Rnews/Rnews_2003-3.pdf`.

Jan De Leeuw, Frederic Udina, and Michael Greenacre. Reproducible research: the bottom line. Technical report, 2001.

Friedrich Leisch. Sweave: Dynamic generation of statistical reports using literate data analysis, 2002.

John Macfarlane. Pandoc, 2014. URL `http://johnmacfarlane.net/pandoc/`.

Jeroen Ooms, Duncan Temple Lang, and Lloyd Hilaiel. *jsonlite: A Robust, High Performance JSON Parser and Generator for R*, 2014. URL `http://CRAN.R-project.org/package=jsonlite`. R package version 0.9.12.

Open Knowledge. AnnotateIt JavaScript library, 2014. URL `http://ckeditor.com/`.

Jonathan Robie, Don Chamberlin, Michael Dyck, and John Snelson. XML Path Language (XPath) 3.0, 2014. URL `http://www.w3.org/TR/xpath-30/`.

RStudio. R Markdown, 2014. URL `http://rmarkdown.rstudio.com/`.

David J. Scott. *hwriterPlus: Extending the hwriter Package*, 2011. URL `https://r-forge.r-project.org/R/?group_id=1269`. R package version 1.0.

Yihui Xie et al. *knitr: A general-purpose package for dynamic report generation in R*, 2013. URL `http://yihui.name/knitr/`. R package version 1.5.