

# Can Web Technologies Help R Generate Print Quality Graphics?



NOTE: Image to be updated with better example

CompSci 791: Research Paper [Working version]

Kane Cullimore

September 22, 2020

**Abstract**

To be continued...

# Contents

<b>1</b>	<b>New Structure</b>	<b>5</b>
1.1	Front Cover . . . . .	5
1.2	Introduction . . . . .	5
1.3	Problem Definition . . . . .	5
1.4	Solution . . . . .	6
<b>2</b>	<b>Introduction</b>	<b>7</b>
<b>3</b>	<b>Problem Definition</b>	<b>8</b>
3.1	Print Quality Graphics . . . . .	8
3.2	Existing R Workflows . . . . .	10
<b>4</b>	<b>Solution Definition</b>	<b>12</b>
4.1	Design & Requirements . . . . .	12
4.2	Web Technologies . . . . .	15
<b>5</b>	<b>layoutEngine</b>	<b>16</b>
5.1	Overview . . . . .	16
5.2	Design . . . . .	16
5.3	Benefits . . . . .	16
5.4	Drawbacks . . . . .	16
<b>6</b>	<b>DOM Backend</b>	<b>17</b>
6.1	Overview . . . . .	17
6.2	Design . . . . .	17
6.3	Examples . . . . .	24
6.4	Benefits . . . . .	26
6.5	Drawbacks . . . . .	26
<b>7</b>	<b>RSelenium Backend</b>	<b>27</b>
7.1	Overview . . . . .	27
7.2	Design . . . . .	27
7.3	Examples . . . . .	27
7.4	Benefits . . . . .	27
7.5	Drawbacks . . . . .	27

<b>8 NodeJS Backend</b>	<b>28</b>
8.1 Overview . . . . .	28
8.2 Design . . . . .	28
8.3 Examples . . . . .	28
8.4 Benefits . . . . .	28
8.5 Drawbacks . . . . .	28
<b>9 Comparison of Workflows</b>	<b>29</b>
<b>10 Recommendations and Future Work</b>	<b>30</b>
<b>A Appendix</b>	<b>31</b>
A.1 Development Environment . . . . .	31
A.2 Emacs within Docker Container . . . . .	34
A.3 Report Editing Process . . . . .	34
A.4 Org-mode examples . . . . .	35
<b>B References</b>	<b>39</b>

# 1 New Structure

## 1.1 Front Cover

- Include the primary graphic to showcase the ability of using layoutEngine
- Maybe a print of the graphic on an open page to emphasize its a “print”
- Should showcase the primary benefits of what you can’t do within R

## 1.2 Introduction

- Include explanation of the document structure and flow of the narrative

## 1.3 Problem Definition

- Narrow into the specific problem at hand quickly
  - Print Quality Graphics
  - From within R (i.e. don’t cover latex and adobe, etc)
- Get a list of specific things that can’t be done in R
  - Typesetting / Layout of text wrapping
  - Font style control
  -
- What is it?
  - Integration of the image within other content that is accessible programmatically
  - Not just an embedded graphic
  - HTML knows about the interior of the R graphic and is NOT just a dumb blob
- What are some available options in R that we know about (not exhaustive)
  - GGText
  - Grid Layout
  - Patchwork (arrangements of plots)

NOTES:

- Avoid trying to prove we can’t do what we’re doing in another package
- Express our focus as being “different” from other approaches (not “better”):
  - Trying to get at all of HTML with as little work as possible
  - Take advantage of what already exists (i.e. to generate HTML)
- Don’t spend too much time talking about layoutEngine and backends

- Use a comparison chart at the end of the paper
- Don't explain web technology choice in detail (just that it can do these things so we chose it) then move on
- My work (or solution) improves on the Cons of the layoutengine and its backends
- "Solution" is the layoutEngine but it is still being developed
- Examples
  - Should be a series of simple examples to talk about each pieces that extends what R can do
  - Should be 1 (or 2 max) complex examples that showcases the full range of abilities

## 1.4 Solution

- HTML/CSS/JS can do these things so we chose these technologies
- Want to not just wrap a single piece of functionality but get access to the full range of those technologies
- Pros
  - Access to a huge amount of functionality of the web technology stack
  - Web Tech is a vibrant community
  - Browsers are extremely sophisticated and competitively being enhanced each year
  - Take advantage of the large variety of packages and methods that currently generate HTML
    - \* Knitr (markdown?)
    - \* xtable
    - \* Others?
- Cons
  - Have to learn and write in HTML/CSS/JS
  - Security layer around using a browser
  -
- layoutEngine
  -
- Backends
  - DOM
  - PhantomJS
  - CSSBox
- Graphic of workflow
  - Could show phases with colors
  - Could show the process coming back into R as opposed to exiting immediately
- Explain the Cons and why a new solution is being sought

## 2 Introduction

The **R programming language** is a popular open-source tool used to perform statistical analysis. The language has many reliable libraries and methods to perform sophisticated statistical techniques with relative ease.

Statistical analyses often benefit from the use of graphical representation of data to communicate its complexities. Several niche use-cases are arising which require unique graphical capabilities since R has grown in popularity [ref]. R base graphics has long supported high quality graphical output that is accurate and visually effective at communicating complex information. However, it intentionally does not support the needs of niche applications which then falls to the open source community.

The publishing industry has a long history of specifying unique and complex requirements for incorporated graphics. The fields of typography, typesetting and graphic design have their roots in the first movable type printing from the 15<sup>th</sup> century. Modern graphical output requires precise control of (1) content specification, (2) document layout, (3) visual rendering, and (4) output resolution and file format.

The R language is regularly used in analyses whose output appears in widely distributed publications. The statistics and baseline graphical plots are handled by R. The graphics are then post-processed with external applications such as L<sup>A</sup>T<sub>E</sub>X, Adobe or similar layout and graphical software to adhere to the publisher's requirements. The final result might be satisfactory however the workflow is far from efficient and often relies on the expertise of others.

The workflows produce acceptable outputs but rely on tools that fall outside of R. This requires developers to either hand-off work to other specialists or to learn these external tools themselves.

A primary feature being explored is whether print quality graphics can be generated from within the R ecosystem. More specifically, this research focuses on how web technologies might help R users generate these print quality graphics for direct use within the publishing industry. A primary focus is placed on the ease of use for existing R users so as to extend their value directly into the published document.

The research looks several existing R packages that are in various states of development [Ref]. These include the [Layout Engine] and the [DOM] packages. It also includes preliminary development of extensions to these packages. These packages explore the use of several popular web technologies.

The research finds there is promising improvements to be found by adopting web technology tools and techniques. [More details to follow here...](#)

## 3 Problem Definition

Often statistical analyses lead to published communication of the process and results. This might be included within websites, books, journals or television. Certain platforms with large audiences will go to greater lengths to include higher quality graphical output. Efforts focus on many aspects of the output format. Some of these might include adherence to organizational branding or improving the effectiveness of communication and reader engagement with the use of certain aesthetic features.

As already mentioned, the functionality of the R programming language is squarely focused on the rigors of statistical analysis and the accurate and faithful representation of data. As a result of this focus, many R users within industries that generate published work have been reliant on downstream tools to post-process graphics.

This subset of R users are the focus of this research.

The primary requirements for generating print quality graphics being explored include:

1. Generate the final graphic from within the R ecosystem
2. Simplify the workflow
3. Use **web technologies** as the platform to extend R's graphical capabilities

### 3.1 Print Quality Graphics

There are several aspects of generating print quality graphics. They can be categorized into the following groups:

1. Content (fonts, colors, icons, etc)
2. Layout
3. Rendering
4. Image format and resolution

#### 3.1.1 Content Specification

The publishing world relies heavily on customized visual content to help communicate in various ways. Colors are altered to improve the readability for the color-blind or for other environments such as a bright display screen. The selection and styling of font can be an art all unto itself. Icons and graphics can be added to draw attention or better communicate the characteristics of the data.

The standard R graphics output are well designed for an audience whom are primarily interested in the data alone. As an example, there are standardized sets of data-point icons [ref and provide figure] and a basic coverage of font options. However, to meet the needs of the publishing industry a much more flexible system is needed to allow custom user-defined options to be used within the graphics. **Provide example...**

- Custom fonts

- Mathematical equations
- Data point icons

### 3.1.2 Document Layout

The layout of content within books, magazines and websites is an especially important aspect of generating print quality graphics. The standards of typesetting have a long history and many general rules and guidelines exist within the publishing industry [ref].

Traditional typesetting often arranges content within a grid system that adheres to well-known relationships with other components on the page. Graphic design professionals will often make slight tweaks that are difficult to prescribe within a predefined set of layout options. Rather, a flexible system that allows full control of all content positioning axes is needed.

- Fine typesetting controls
- others(?)

Include diagram of what the typesetting might include (similar to Rahlfs book (Chp 4))

### 3.1.3 Visual Rendering

To meet the rich aesthetics of the publishing industry a powerful rendering engine is needed to cope with a variety of visual requirements. These include fine control of colour, gradients, resolution, scale, etc.

R has very high quality rendering engine in its baseline configuration however there are a number of features it does not support.

- Color gradients
- others(?)

### 3.1.4 File Resolution and Format

R has a strong capability in its support of various file formats. There are several file format types which are preferred by users generating print quality graphics. Such file types help maintain quality, information preservation, consistency across the industry and performance (file size, read/write speeds, etc).

- Vector: PDF and PostScript
- Raster: PNG, JPEG and TIFF
- XML: SVG
- Web: HTML, CSS and JavaScript
- Color gamut

## 3.2 Existing R Workflows

Extensions have made the output graphics reach a broader variety of plot-types and effectiveness of the visualization [ref Paul's book with respect to lattice and ggplot]. However, most often the final published versions have undergone typesetting within an external post-processing step.

Despite the vast number of available R extensions there are few that improve the workflow for generating print quality graphics.

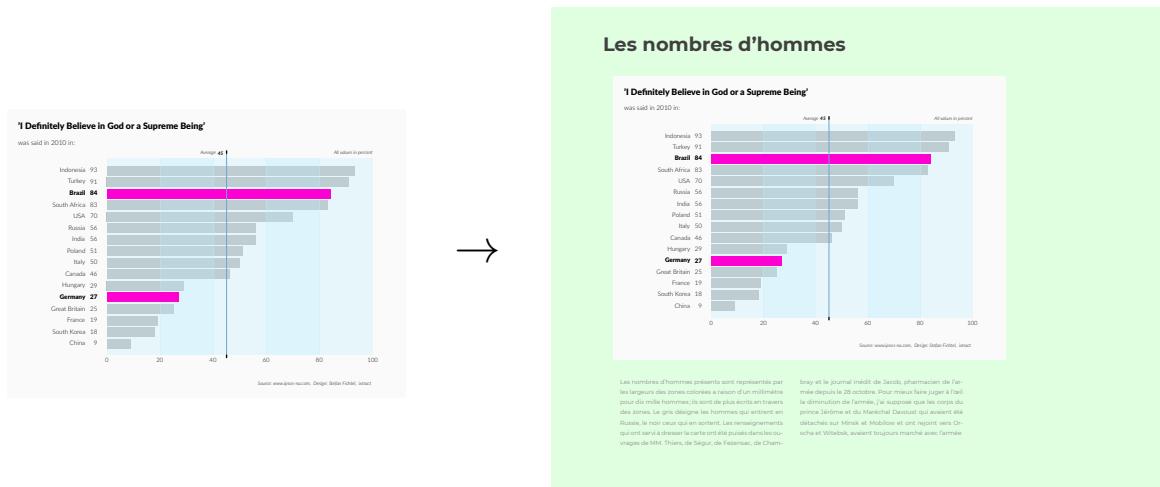
There are several which extend R's graphical capabilities but none that capture the necessary precision needed in publications. For example, R can be used to drive an interactive web-based application with the Shiny package [ref]. A variety of aesthetically pleasing standard plots can be quickly generated with the ggplot2 package [ref]. Several smaller packages also provide helper methods to enhance color selection [ref] or modify output file formats to make downstream work more smooth [ref - svg, grid?, etc?].

The current industry employs various tools and methods to achieve these outcomes.  $\text{\LaTeX}$  is a very popular method for publishing documents. The Adobe suite of tools are used for a range multi-media mediums such as websites and large advertisement prints. Several others exist but are not introduced.

### 3.2.1 $\text{\LaTeX}$ Environment

$\text{\LaTeX}$  is a widely used and rich typesetting programming language that can be used to publish text books, journals, reproducible research [ref ]. The following example shows how a plot generated within R can be transformed into a much richer document.

NOTE: Just a placeholder. Will use a better example and reformat to better show workflow.



### 3.2.2 Graphic Design Software

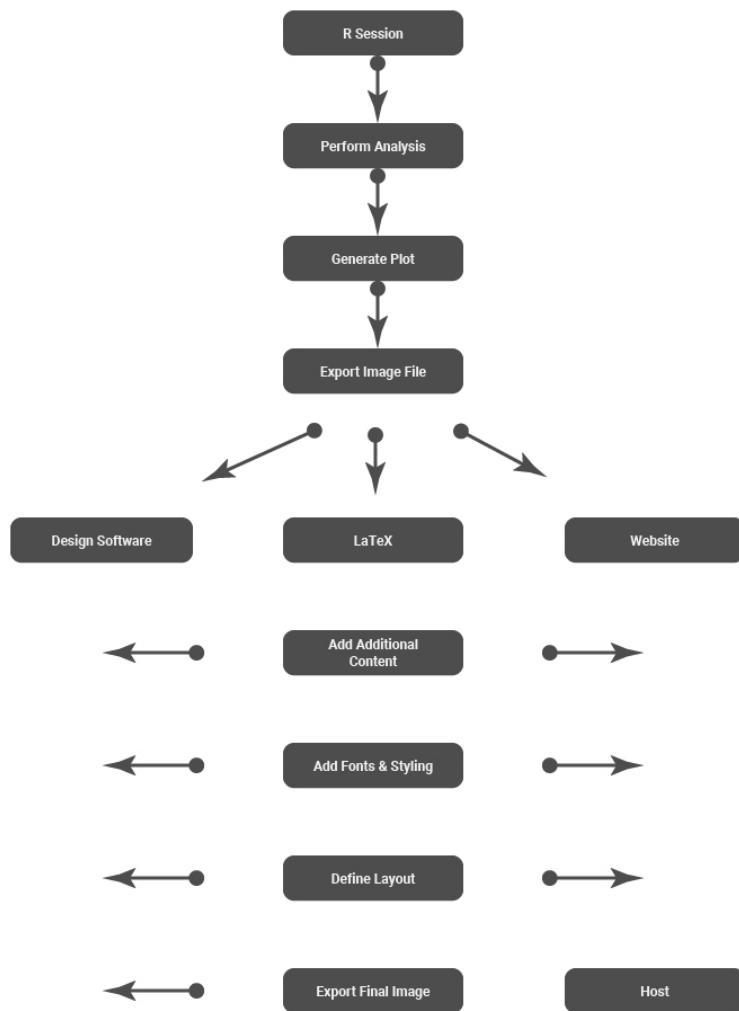
There are many graphic design and publishing software packages available. Most are designed for the use by artistic professionals with Graphical User Interfaces (GUI) controls. Many also have the ability to be extended via macros or other methods to programmatically control the work flow. **Adobe** [ref] offers a suite of software and is a market leader in this field.

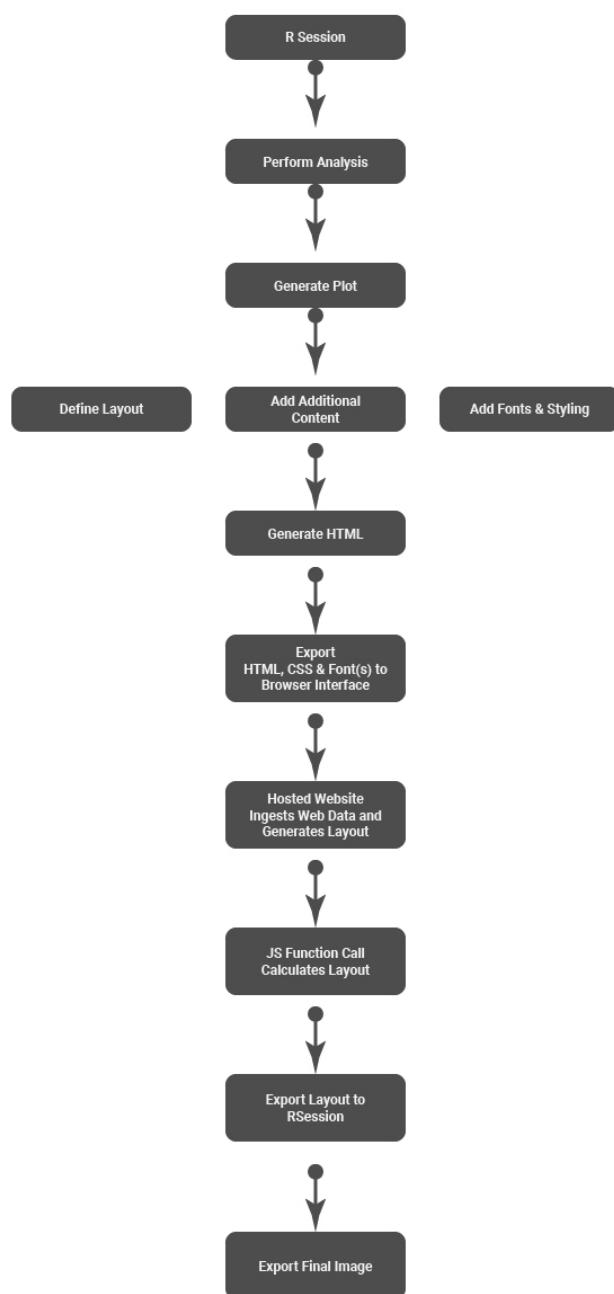
NOTE: Just a placeholder. Will include an example of using Illustrator and InDesign

## 4 Solution Definition

### 4.1 Design & Requirements

- Send HTML, CSS, JS and font description to browser to render/layout
- After browser has loaded the data some JavaScript calculates and returns layout dimensions to the R session (and perhaps other styling info)
- The browser should have an ability to use the host machine default browser or run in the background on a headless browser
- Easy to use on all major operating systems (Windows, macOS and Linux)
- Minimal installation requirements
- Access via any modern web browser on the host machine
- Lightweight
- Performance





## 4.2 Web Technologies

### 4.2.1 State of Modern Browsers

- Modern browsers are capable graphical display engines
  - Active specification and developer groups
  - Growing improvements in browser capabilities
  - Layout engines that mostly adhere to web standards
    - \* Gecko (FireFox)
      - WebKit (Apple Safari)
      - Blink (Google Chromium and Chrome)
      - EdgeHTML (Microsoft)
      - Presto (Opera)
    - Large investment activity due to amount of supported commerce
  - Browsers widely available on all user devices
    - PCs and portable devices
    - Users comfortable and familiar within the browser environment

### 4.2.2 HTML Standard

The specification for the HTML standard is constantly evolving. This is generally a good thing as new technologies are continuously evolving and improving the web experience. However, the continuous updates creates a moving target to developers of both the web browsers as well as websites.

The HTML specification is controlled by the World Wide Web Consortium (W3C) [ref]. Various browsers support most of the newly available definitions being released however not all are in sync. The browsers that typically support the latest specifications sooner are Mozilla Firefox [ref] and Google Chrome [ref]. Apple Safari [ref] is often not far behind. Microsoft has notoriously been sluggish in their adoption of the latest specification with its Internet Explorer browser which was only updated along with its Windows operating system. Most recently Microsoft's superseeding browser, Edge [ref], is much better in its support.

For the use of development it is recommended that either Firefox or Chrome are used with DOM. The discussion of potential updates to DOM will take into consideration the need to be compatible with Safari and Edge into the future.

## 5 layoutEngine

### 5.1 Overview

#### 5.1.1 Links

- layoutEngine Overview
- GitHub Repository

### 5.2 Design

Technologies

.

Dependent Packages:

- The htmltools R package (for )
- The xml2 R package (for )
- The extrafont R package (for )
- The gdtools R package (for )

### 5.3 Benefits

### 5.4 Drawbacks

## 6 DOM Backend

### 6.1 Overview

The DOM package is built with several existing packages described below. The package intends to meet the solution design described above. The package has been through several development cycles at the time of this report. These improvements have focused on...(capture main items).

The package is not focused on generating interactive web applications such as the `shiny` package. Rather, it is designed to incorporate web technologies to improve desktop publishing, typesetting and layout functionality (see `LATEX` stack, Adobe, others??). (Reword => Intended as generating CODE interfaces (not graphical interfaces)...(I need to understand this better))

In the same way a web-technology stack that goes about this with HTML, SVG and JavaScript the DOM package would intend to build a HTML, SVG and R stack (? reword...).

#### 6.1.1 Links

- DOM Version 0.4
- GitHub Repository
- 

### 6.2 Design

A primary goal of the DOM package is to perform typesetting post-processing tasks while remaining in the R environment. This being as opposed to taking an unfinished graphic output as a **pdf** file into other environments such as `LATEX` or Adobe tools.

DOM aims to achieve this by employing the web standards set in HTML, CSS and JavaScript. The R plot would be sent along with such code into a web browser which would render the final document.

The benefit of such a work-flow lies in taking advantage of one of the richest consumer focused programming environments that exists. **Find some reference pointing to the popularity and efficacy of this.**

Technologies

- Web Sockets
- JavaScript DOM Scripting

Dependent Packages:

- The `httpuv` R package (for websockets)
- The `jsonlite` R package (for JSON messages on websockets)
- The `css-select-generator` JS library (for CSS selectors)
- The `bowser` JS library (for identifying the browser)

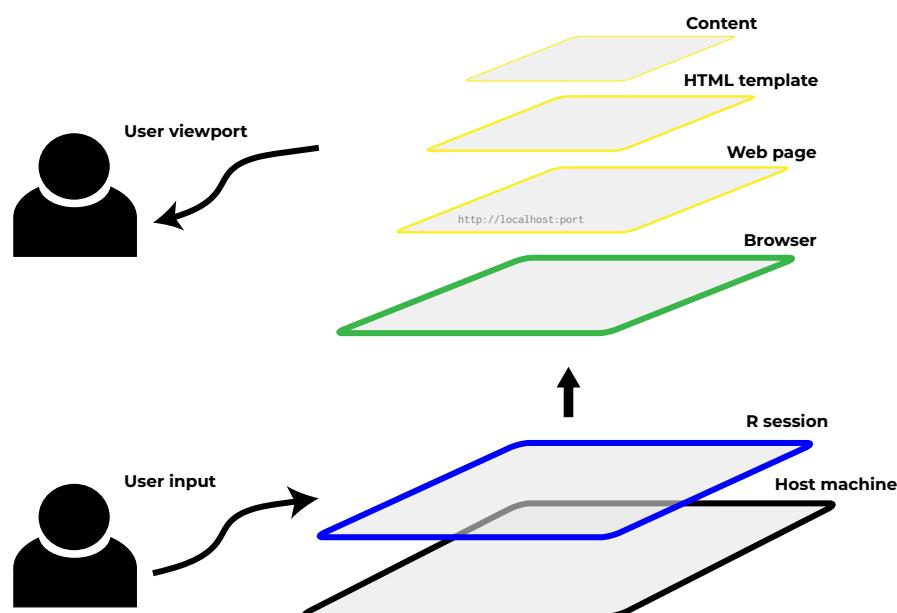
### 6.2.1 Use-case scenarios

Need diagram to show various paths to use DOM

- Some plot -> gridSVG -> SVG -> Render in browser or phantomJS -> code for layout or print
- data frame -> xtable -> HTML -> Render in browser or phantomJS -> code for layout or print
- Review `layoutEngine` which will take HTML code and incorporate back into R graphics

### 6.2.2 How it Works

- An R session will act as a web server by hosting a web page within the local network (i.e. 127.0.0.1:port a.k.a. localhost:port)
- The web page is only accessible on the local machine and not intended to be publicly hosted
- A user will then interactively generate a graphical output within the browser window essentially using the browser window as their *canvas* which provides the visual feedback
  - This will be achieved by first setting up an initial html template based on predetermined needs (width/height, colour schemes(?), layout(?), etc)
  - The graphic components are then built with packets of web data by pushing it from the server (e.g. laying down the axes and scales first, then the data points and then the text boxes)
  - Information from the browser will be sent back to the server since some component data might depend on browser information such as knowing exact size or locations of the page elements relative to each other
  - It is possible (as the package evolves) the user might interact with the web page and therefore send data back to the browser via requests (e.g. changing an axis range or exporting) however, since the same user has control of the browser via the server this secondary control avenue doesn't seem necessary
- Once the graphic is completed the user might export this as a file either as web data (html, xml, etc) or as a rendered image (pdf, png, etc)

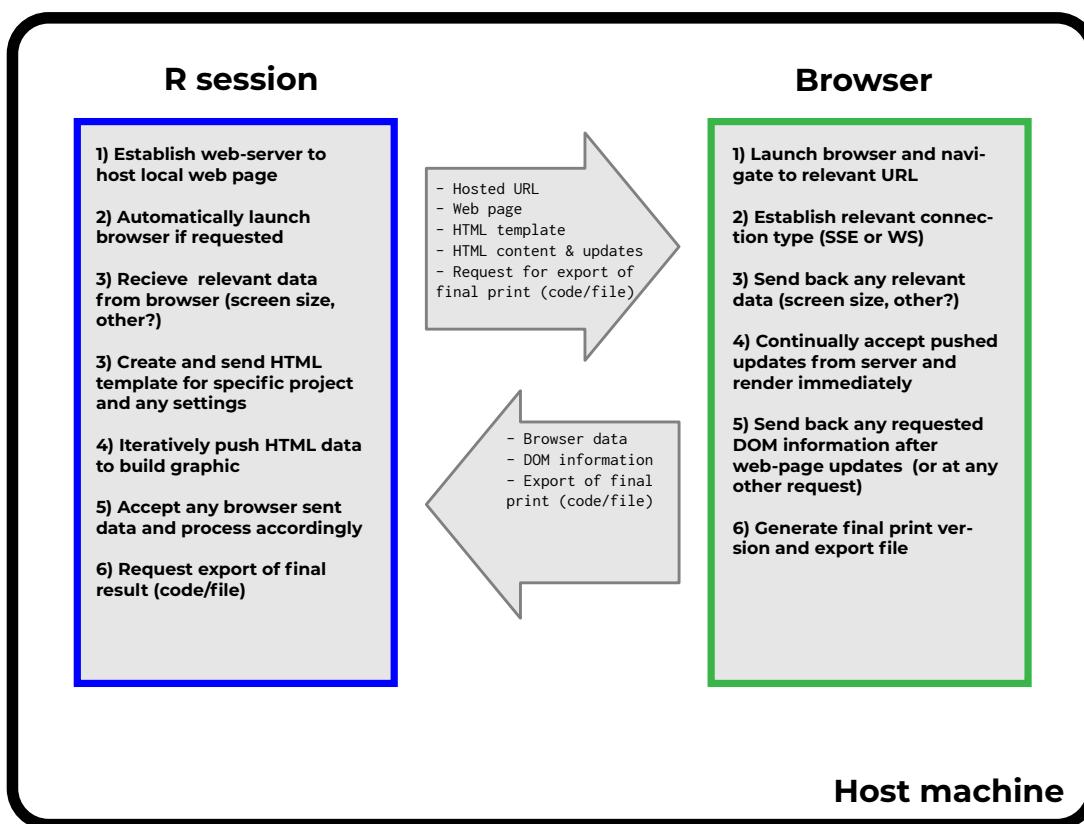


### 6.2.3 Technical Deep Dive

#### 1. Technical requirements

The general requirements to support this solution design are:

- (a) The R session must create a non-blocking web server to locally host the web page as a background service while allowing the user to interact with it
- (b) Various html templates and accompanying arguments (e.g. screen width/height) must be able to be selected and loaded upon initialization of the web page
- (c) A browser on the local network will launch to the web page with a specified URL (either launched automatically or manually)
- (d) The server will iteratively push updates to the web page description
- (e) The browser will automatically update its rendered version of the web page upon any changes pushed by the server
- (f) The browser will be able to send data back to the server either by (1) request of the server (i.e. a server query) or (2) in response to any update pushed to the browser
- (g) **ON HOLD:** User can send requests to the server via a browser interface (forms, buttons, sliders, etc)
- (h) The final result must be able to be exported as (1) code and (2) a rendered image



## 2. R Package Design

The DOM package is built with the following packages:

- (a) `methods`: S4 class system
- (b) `utils`: Basic methods for launching default browser, capturing output and extracting the package version
- (c) `httpuv`: The HTTP and WebSocket server functionality
- (d) `jsonlite`: To transfer data between R and the browser via JSON
- (e) `whisker`: A HTML template building system
- (f) `Rook`: A HTTP message specification that `httpuv` adheres to

When DOM is loaded into R several parameters are written to the global options from which R can access programatically (i.e. review via `.Options`)

- `DOM.client` includes the basic functions to launch the application via `httpuv`, run the browser session and kill the server.
- `DOM.debug` is a flag for...
- `DOM.limit` is the maximum number of concurrent server sessions to create (default to 5)
- `DOM.width` is the DOM.window width (default at 800 px)
- `DOM.length` is the DOM.window length (default at 600 px)

### 3. Inner Workings

The DOM package has a very simple process for establishing a blank (or simple) HTML page. The `htmlPage()` method used to achieve this [ref] abstracts several processes from the user. The following section breaks down what is happening under the hood.

The DOM package has a very simple process for establishing a blank (or simple) HTML page. The `htmlPage()` method used to achieve this [ref] abstracts several processes from the user. The following section breaks down what is happening under the hood.

A basic web page is defined with a single element in the form of a paragraph tag with some text; "`<p>Some text lies here...</p>`". The DOM package initiates this simple web page with the following commands. Although this is small amount of code there is a lot going on in the background.

```
library(DOM)
text <- "<p>Some text lies here...</p>"
p <- htmlPage(text)
```

When the DOM package is loaded there are several objects loaded into the global options list. These are defined in the previous section. The `DOM.client` object is of special interest here since it defines a `app`, `run` and a `kill` method for use with the server calls.

```
library(DOM) ## Loads methods and generates 5 objects within the global options
## 1) DOM.client with 3 methods
## $app: the primary function used to define the server logic when the web page is
##       launched
## $run: the function that is called to launch the browser on the host machine
## $kill: the function called to clean up when the server is shut down
## 2) DOM.debug is set to FALSE
## 3) DOM.limit is set to 5
## 4) DOM.width is set to 800
## 5) DOM.length is set to 600
```

The `htmlPage()` method is run with the following default arguments. This function is defined within the `Page.R` file.

```
text <- "<p>Some text lies here...</p>"  
p <- htmlPage(  
  html=text, ## Body of web page at initiation  
  head="", ## Arguments to place into the HTML header  
  host="127.0.0.1", ## The URL address for localhost  
  client=getOption("DOM.client") ## Default definition of the server logic  
)
```

Once the `htmlPage()` function is called several things happen in the background. Most notably is a list of hosted pages and responses is created to keep track of them. Therefore each page and response associated with a page has a pointer object to identify it. The `getPageID()` function is defined within the `Page.R` file while the `getResponseID()` and `addRequest()` functions are defined within the `DOM.R` file.

```
## Inside of the htmlPage() call the following steps are taken  
pageID <- getPageID() ## Assigns an integer value within a list of page IDs  
tag <- getResponseID() ## Assigns an interger which is converted to a character  
## The following function call adds the new tag to the list of open requests  
addRequest(  
  tag=tag ## New tag values  
  async=FALSE ## Whether the request should block (syncronous) or not  
  ↪  asynchronous)  
  callback=NULL ## If the request is asynchronous then what callback to use  
  returnType="NULL" ## ??  
  pageID=pageID ## The pageID the request is assoiated with  
)
```

Next the `htmlPage()` function calls the `startDOMServer()` function which is defined within the `Page.R` file. This function takes the arguments defined so far and sends it to `httpuv::startServer()` which is the primary mechanism to launch the server and host the web page.

```
## The arguments are passed from htmlPage() and the newly created page and response  
→  ids  
startDOMServer(  
  pageID=pageID, ## id assigned to the server instance  
  host=host, ## URL to launch the web page with  
  app=client$app, ## from the global options  
  port=NULL, ## Currently the port # is randomly assigned at the next step  
  body=html, ## html content to pass along  
  head=head, ## any headers to add to the web page  
  tag=tag ## the request id  
)
```

Once the server has been successfully created via the `httpuv::startServer()` function the following steps are taken.

```
## 1) The randomly assigned port # is captured in a private variable
port <- pageInfo(pageID)$port
## 2) The defined client$run method is called
## By default a browser is opened and navigated to the specified URL
client$run(
  url=paste0("http://", host, ":", port, "/")
  host=host,
  port=port,
  tag=tag
)
## 3) The active response is added to a list to track when its resolved
waitForRequest(
  tag=tag,
  limit=getOption("DOM.limit"), ## Default value for number of open requests
  onTimeOut=function() closePage(pageID) ## Used to prevent indefinite hanging
)
```

To be continued...

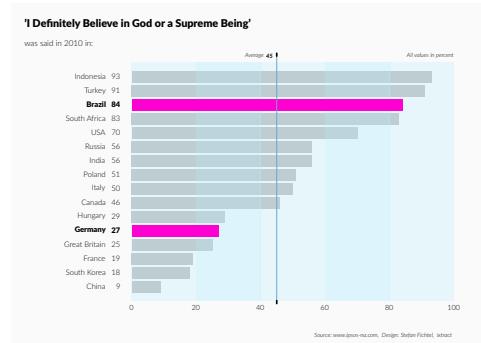
## 6.3 Examples

A series of examples are provided to demonstrate how DOM can be used to perform the various aspects of generating print quality graphics. All are intended on being reproducible within the Docker environment explain in the appendix [ref].

### 6.3.1 A simple overview

The following overview describes a simple workflow to use R to create a plot and then add some styling and additional components within the browser.

1. Generate a plot within R using a grid based package (`grid`, `lattice` or `ggplot2`)
2. Convert the plot to **SVG** with `gridSVG`
3. Initiate a webpage with DOM and navigate to the URL with a browser to initiate the WebSocket connection
4. a;lsdkfj;lasdkfj;alsdkfj;alsdfkj



### 6.3.2 Content

Ideas:

- Custom icons as data points

.

### 6.3.3 Layout

Ideas:

- 2 or 3 column layout over plot
  - Various justified
  - with CSS-grid
- Generate different plot layouts for various screen (or document) sizes
  - with CSS-grid and media queries

### 6.3.4 Rendering

Ideas:

- Gradient fill along bars
  - with gridSVG and push across
  - with CSS styling
  -

### 6.3.5 Exporting

Ideas:

- Using

## **6.4 Benefits**

## **6.5 Drawbacks**

## 7 RSelenium Backend

### 7.1 Overview

#### 7.1.1 Links

- GitHub Repository
- RSelenium R Package
- Selenium Server

### 7.2 Design

### 7.3 Examples

#### 7.3.1 Overview

#### 7.3.2 Content

#### 7.3.3 Layout

#### 7.3.4 Rendering

#### 7.3.5 Exporting

### 7.4 Benefits

### 7.5 Drawbacks

## 8 NodeJS Backend

### 8.1 Overview

#### 8.1.1 Links

- GitHub Repository

### 8.2 Design

### 8.3 Examples

#### 8.3.1 Overview

#### 8.3.2 Content

#### 8.3.3 Layout

#### 8.3.4 Rendering

#### 8.3.5 Exporting

### 8.4 Benefits

### 8.5 Drawbacks

## 9 Comparison of Workflows

## 10 Recommendations and Future Work

# A Appendix

## A.1 Development Environment

A single Docker container is used to perform research, experimentation, R package development and documentation. This environment was chosen to easily share the development content with others for collaboration and feedback. It will also ensure that any future return to this research can be resurrected with a working code-base independent of software changes.

The report and R development have been performed within Emacs and ESS environment inside of the Docker container. The report is written within the Emacs org-mode markdown language which abstracts some  $\text{\LaTeX}$  syntax while also providing literate programming options which are more flexible than generic markdown or Rmarkdown.

Some basic Docker and Emacs commands are provided to walk the user through some of aspects of the build and editing processes.

### A.1.1 Docker container description

Overview: The Docker container is publicly available on Docker Hub with the following image name **kculldom\_r**. The container is built from the Ubuntu 18.04 image and has R 3.6.1 and Emacs 26.3 installed. The container has been configured to run Emacs in its GUI environment on the host machine.

User and Home Directory: The user is logged in as a sudo-user with `/home/user/` as the `$HOME` directory. The sudo password is “password.” The working directory is `/project/` which both the shell and Emacs will initialize into.

Directory Organization: The project is organized into 2 main directories which are each linked to github repositories. Directory 1 is the forked DOM package which is extended per this research. Directory 2 is the research effort which contains the experimental code and files used to create this report. Both directories should be created and maintained on the host machine and then connected to the Docker container with a **bind mount** option when the container is running.

Directory Hierarchy:

```
# Emacs configuration files  
/home/user/.emacs.d/  
# Github repository for R DOM package  
/project/DOM/  
# Github repository for research documentation  
./.../..  
# Experimental code for research  
./.../..../code  
# Files to generate report  
./.../..../paper  
# Miscellaneous files such as the Dockerfile, Fonts, Images, etc.  
./.../..../resources
```

### A.1.2 Host setup and Docker run instructions

The following instructions are provided to recreate the development environment. This has only been tested from within a host machine running Ubuntu 18.04 but is assumed to be compatible with other Debian derivatives. The Dockerfile used to build the container is supplied which should enable the necessary tweaks to support other host machine environments.

1 - Ensure docker and git are installed and correctly configured to download the docker image and clone repositories.

2 - Obtain access to the private github repository, [https://github.com/kcullimore/DOM\\_research](https://github.com/kcullimore/DOM_research), for the research content by emailing myself at <kane@ia.house>. The DOM package, <https://github.com/kcullimore/DOM>, is publicly available.

3 - Create a working directory on the host machine and clone each github repositories as shown below:

```
## Example folder structure
mkdir -p ~/project/{DOM,research}
## Clone repositories
git clone git@github.com:kcullimore/DOM.git ~/project/DOM
git clone git@github.com:kcullimore/DOM_research.git ~/project/research
```

4 - Run the following commands within the host terminal but first ensure the target directories are correct per Step 3 (i.e. verify source= **/home/\$USER/proj...** is correct).

```
## Open host machine devices to container (only need to do once per reboot)
xhost +local:
## Run docker container linked to directories with bind-mount -----
docker run --rm -it \
    --network host \
    --env DISPLAY=unix$DISPLAY \
    --volume /tmp/.X11-unix:/tmp/.X11-unix \
    --mount type=bind,source=/home/$USER/project/DOM/,target=/project/DOM/ \
    --mount type=bind,source=/home/$USER/project/DOM_research/,target=../../.. \
    --name dom-test \
    kcull/dom_r:latest
```

5 - Once the docker container is up and running verify folder structure has correctly mapped the host directories.

6 - Open Emacs in the container's terminal: \$ Emacs. The host should launch Emacs in its GUI form (i.e. not within the shell). If this doesn't occur verify steps 4 were followed thoroughly (NOTE: After reboot the display device will have to be provided access again with the {\*xhost + local : command*}).

7 - From within Emacs perform the following operations to open and recreate the current report

- Opens Treemacs with M-0
- Open folder structure to ./.../paper/ with Tab-Enter or Mouse
- Open org-mode markdown file /paper\_working.org with Enter or Mouse double-click
- Make some edits to the file and save with C-x C-s
- Launch Export Dispatcher menu with C-c C-e
- Create new PDF file with C-l C-o

8 - The PDF should have opened automatically which you can scroll through with arrow keys or the mouse scroll wheel. Use q key to minimize the PDF buffer.

9 - Close Emacs with C-x C-c and exit the container by typing exit at the terminal.

10 - Navigate to the project directory on the host machine and verify the new PDF and edited org-mode file were correctly saved.

11 - If the above worked the project appears to be correctly established on the host machine.

## A.2 Emacs within Docker Container

### A.2.1 Useful report editing commands

#### Emacs Terminology

- **buffer:** 'Screen' or 'window' user operates within
- **marking:** Highlighting region of window

Often used commands can be found at <https://www.gnu.org/software/emacs/refcards/pdf/refcard.pdf>.

#### Customized keybindings

- Open emacs configuration file with C-c e (Emacs must be restarted for changes)
- Expand all nested/hidden text within \*.org file with Shift-Tab Shift-Tab Shift-Tab
- Copy, cut and paste with standard keybindings per **Cua Mode**
- Switch visual line wrap with M-9
- Switch to truncate long-line view with M-8
- Enter/Exit rectangle edit mode with C-^
- Enter/Exit multi-edit mode by highlighting word and then C-u
- Auto-indent R script (via ESS) by highlighting buffer with C-x h and then C-M-}

#### Document Export

When a PDF version of the document is produced a standard  $\text{\TeX}$  file (\*.tex) is also produced after transpilation. This  $\text{\TeX}$  file can be edited and used with a standard  $\text{\LaTeX}$  command: `latex report.tex`.

To be continued...

## A.3 Report Editing Process

To be continued...

## A.4 Org-mode examples

### A.4.1 Font definitions

Using L<sup>A</sup>T<sub>E</sub>X fontspec package [1]

Sans

Internet based applications are an increasingly popular way to communicate and interact with complex data.

Sans italic

*Internet based applications are an increasingly popular way to communicate and interact wtih complex data.*

Sans italic bold

***This might include a business application that assist employees unverstand the current state of the market.***

Serif

It might also include a news website communicating techincal details from a story such census data.

Serif italic

*It might also include a news website communicating techincal details from a story such census data.*

Serif italic bold

***It might also include a news website communicating techincal details from a story such census data.***

Mono type

It might also include a news website communicating techincal details from a story such census data.

Mono Bold type

The quick brown fox 012456789

### A.4.2 Sample R code highlighting

```
##*****10*****20*****30*****40*****50*****60*****70*****80
## Problem 2: START => Optical Illusion Example
##*****10*****20*****30*****40*****50*****60*****70*****80
## Generate pdf file of plot (capture ends with dev.off() below)
pdf("prob-02.pdf", width = 3, height = 6)
## Create theta values for each line segments (i.e. 180 degs / 4 = 45 segments)
## Remove elements in the center of vector (i.e. 80-100 degree section)
theta <- seq(0, pi, length = 45)[-20:26]
## Set parameters to be used in plot() (R = dummy radius, B = slope of lines)
R <- 1
B <- sin(theta) / cos(theta)
## Setup plot space and define coordinate axes (also remove 'edge buffer')
plot.new()
par(mar = c(0.1, 0.1, 0.1, 0.1))
plot.window(xlim = c(-R, R), ylim = c(-R, R), asp = 1)
## Create the black line segments
for (i in 1:length(B)) abline(a = 0, b = B[i], lwd = 2)
## Create the 2 red vertical lines
abline(v = c(-R/2, R/2), col = "red", lwd = 4)
## Stop image capture
invisible(dev.off())
##*****10*****20*****30*****40*****50*****60*****70*****80
## Problem 2: END
##*****10*****20*****30*****40*****50*****60*****70*****80
```

#### A.4.3 Sample HTML code highlighting

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta
      name="viewport"
      content="width=device-width, initial-scale=1,
      maximum-scale=1.0, user-scalable=0"
    />
    <!-- favicon -->
  </head>
  <body>
    <title>DOM - Testing Application</title>
    <div id="AppDiv" class="app-div"></div>
  </body>
</html>
```

#### A.4.4 Sample CSS code highlighting

```
.iah-text-Raleway {
  font-family: 'Raleway', sans-serif;
  font-weight: 500;
}

.iah-text-black {
  font-family: 'Roboto Mono', monospace;
  font-weight: 500;
  font-size: 2em;
  overflow-wrap: break-word;
  margin: 10px;
  color: var(--iah-grey-dark);
}
```

#### A.4.5 Sample JavaScript code highlighting

```
var args = []; // Empty array, at first.  
for (var i = 0; i < arguments.length; i++) {  
    args.push(arguments[i])  
} // Now 'args' is an array that holds your arguments.  
  
// ES6 arrow function  
var multiply = (x, y) => { return x * y; };  
  
// Or even simpler  
var multiply = (x, y) => x * y;
```

## B References

### References

[1] package used to manage fonts within xelatex (or luatex) fonts: <http://ctan.math.washington.edu/tex-archive/macros/latex/contrib/fonts/spec/fontspec.pdf>

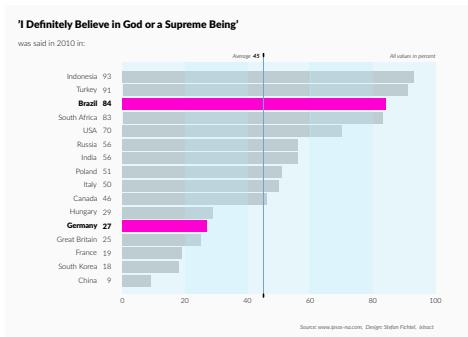


Figure 1: R Plot Output