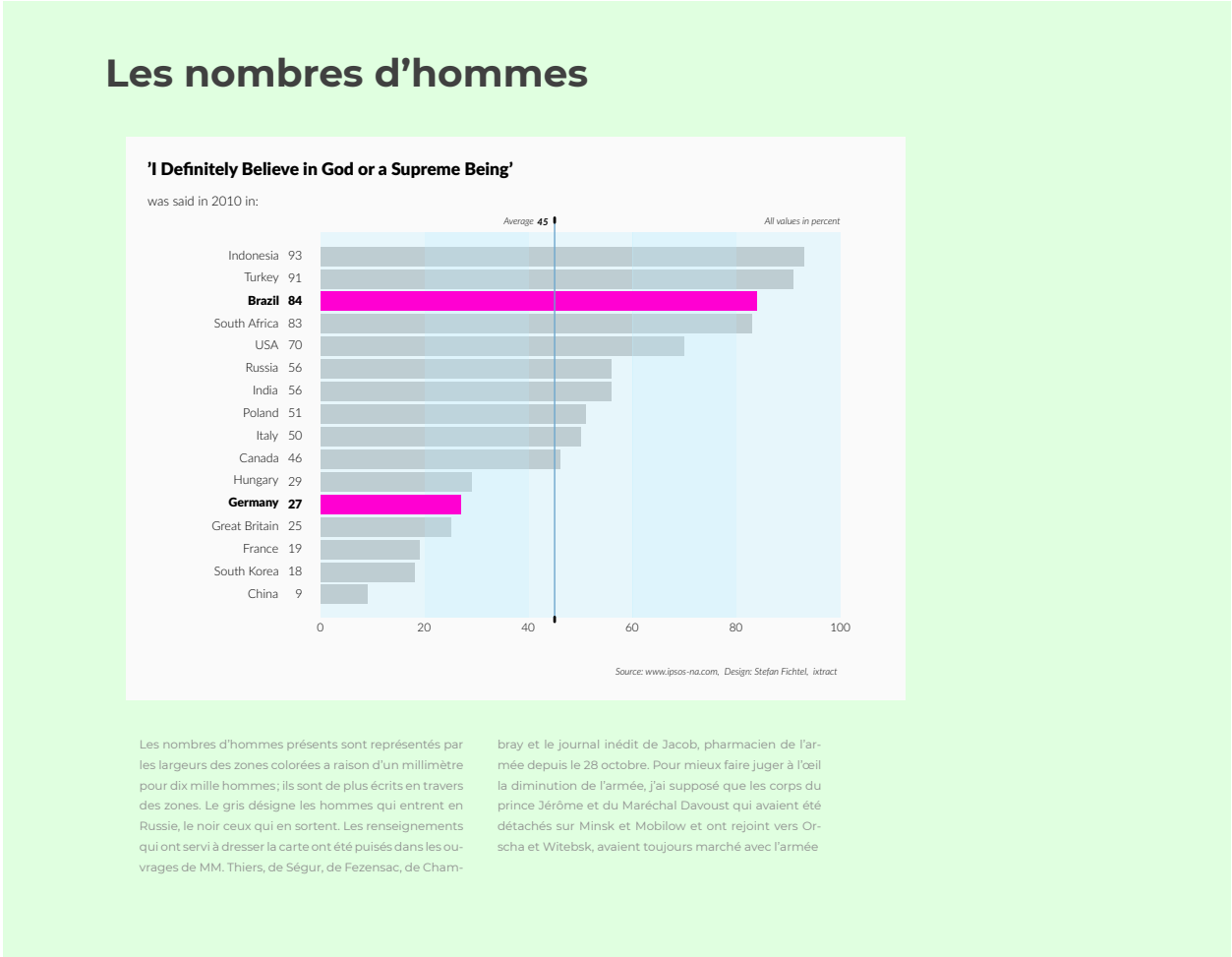
f

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

October 20, 2020

Abstract

Abstract paragraph to cover the following main topics:

- Introduction and problem description
- layoutEngine and its limitations
- RSelenium backend
- NodeJS backend
- Future suggestions

Document Structure

This report is organized in the following way:

- A problem definition is given to explain the core functionality the layoutEngine library is provided
- A brief explanation of how the layoutEngine is intended to be used along with its current shortcomings is covered
- The RSelenium layoutEngine back-end is introduced
- The NodeJS layoutEngine back-end is introduced
- A comparison of each back-end is given to identify areas of advancement
- Finally, a review of the overall layoutEngine approach is given and compared to some existing R extensions

Temporary Notes

The planned size of document:

- Background = 10 pages
- RSelenium Backend = 10 pages
- NodeJS Backend = 10 pages
- Comparisons and Wrap-up = 10 pages

Presentation:

- Time 10 mins with 4 mins Qs
- Slide: Background and What problem it solves
- Quick demo (with backup slide)
- Slide: Diagram of lE system
- Slide: Identification of lE limitations
- Slide: New development
- Demo of latest and greatest examples
- Slide: Qs

Contents

1 Introduction

The **R programming language** is a popular open-source statistical analysis tool. The language has many libraries that support sophisticated statistical techniques. Many of these rely on graphical output to communicate results. A strong appeal of this programming language is the ease at which its **core graphics system** (TODO: add glossary item) generates graphical output that is both accurate and effective at communicating this type of information.

Few open-source alternatives offer an equivalent set of sophisticated statistics married to a flexible and powerful graphics system like R. The Python programming language is a strong competitor. However, its focus is more general purpose with fewer specialized statistics libraries that generate these types of graphics (although it is catching up(??)).

As the R programming language has grown in popularity so to has the number of specialized applications. One such use-case is the incorporation of the statistics based graphics within published articles. While the raw dots and dashes used to generate these graphics is of sufficient digital quality there are several fundamental publishing requirements which are not supported.

2 Problem Description

The publishing industry has a long history dating back to the 15^{th} century when movable type printing was first invented. As the industry evolved within digital platforms it has brought with it a system of long-standing expectations for content. As a result, digital publications often have a myriad of requirements for graphics which are referred to in this report as **print quality graphics**. These requirements include **(1)** writing system and font specifications, **(2)** document layout and typesetting, **(3)** sophisticated content rendering, and **(4)** control over output resolution and file format.

R users have a powerful tool for generating statistics based graphics but they will struggle to support many of these requirements. Examples include (TODO: provide plot examples):

- Using multiple font types in the same graphic
- Embedded tables and text-boxes with wrapped text
- Complex layouts of text and graph components
- TODO: add to or modify list

One existing solution to produce print quality graphics is to modify the R graphical output with external tools such as LaTeX or *Adobe Illustrator*. The user must either be proficient in both environments or have a specialist available to help.

Another solution would rely on several existing R libraries. Such libraries offer bits of this functionality ad hoc. The user would remain within the R ecosystem but might need several libraries depending on the publishing requirements. This modular feature-set composition is the *standard R approach* to extend its functionality.

This research reviews an alternative approach where the print quality graphic is generated from within the R ecosystem with a single general purpose solution. This approach is central to the **layoutEngine** R library which incorporates **web technologies** to extend the functionality of the R graphics system. It is based on the understanding that web technologies and modern browsers have long supported the special needs of the publishing industry. Therefore, if the layoutEngine can successfully utilize the browsers graphics system, it can bring a full-suite of industry leading functionality to R users.

This research does *not* address the relative performance and functionality difference between the layoutEngine approach and the *standard R approach*. Rather, it *first* explores the efficacy of this *layoutEngine approach*, and *second*, attempts to improve upon the implementation of the existing layoutEngine library to address several existing limitations.

3 The layoutEngine R Library

The intent of the **layoutEngine**[TODO: Add to References] R library is to extend R's graphical system by adopting functionality available in web technologies. To achieve this, its core functionality is to act as a high-level interface between R and a web browser and thereby tap into the rich feature set. While the library is available to review via the layoutEngine repository, it is still in development and not yet available in **CRAN**.

3.1 The Standard R Approach

The *standard R approach* to extending its functionality has been paramount to the success of open-source programming languages. Available libraries are loaded into the scope of an environment to gain functionality. If a larger feature set is need then several libraries are loaded as a type of modular system. There are many advantages to this approach and it is a key reason why these languages and user communities have thrived.

Several **available CRAN packages** offer functionality which meet some publishing requirements. Many of these libraries are well executed and perform admirably. The *ggtext* package enables multiple font types to be specified in the same graphic. The *patchwork* package offers a similar functionality specifically for arranging several ggplots with claims of increased simplicity and flexibility. In addition, the base package *grid* has a `layout` function which creates a `Grid` layout object that enables plots from different systems to be arranged together. Many other libraries extend R towards the realm of print quality graphics but no general purpose solution exists at this point in time.[TODO: add all to References] Together they establish that a need does exists for this type of extended functionality for R users.

NOTE: Also references Paul's list per HTML Rendering article

This approach might eventually succeed in offering a general purpose solution to generate print quality graphics. However, several difficulties exist would first need to be overcome. First, the publishing requirements represent an extensive set of functionality. In addition, the variety of graphical output this must operate with is also large. As a result, the task of coordinating a suite of purpose-built libraries that is flexible enough to cover all scenarios would be significant. This is both difficult from both a developer and user perspective due to the large number of functions and objects to handle.

3.2 The layoutEngine Approach

The *layoutEngine approach* differs with the way it extends the functionality of R. It acts as a portal between the graphics system in R and a modern web browser so as to avoid reinventing the wheel. Instead it aims to adopt from an industry that has a long history supporting publishing requirements. This approach bypasses the need to build a complex system from the ground up.

This approach aims to take advantage of preexisting technologies to generate print quality graphics.

First, a user can rely on existing R libraries to generate HTML from R data objects which removes the need to translate between languages. Second, it adopts the sophistication of web technologies and modern web browsers that already support much of publishing industry requirements. In addition, since the R user will have unbridled access to these web technologies, the functionality of the R graphics system would be greatly extended.

Placeholder for Figure: layoutEngine Process

The figure above demonstrates the process of a general use-case. In summary, it starts with a graphic partially defined in R. This is converted to HTML where some additional definition could be added. The browser readable definition is transferred and loaded into a web browser where its layout and rendering engine generate the desired graphic in the browser window. A JavaScript function is then executed to calculate the position of each component on the page. This data is then sent back to R in CSV format where the layoutEngine will convert it to a R readable graphic object. This can then be displayed in the R graphics window or sent directly to an image file.

3.3 Solution Design

The layoutEngine solution is designed as a two-component system. The layoutEngine **primary library** is configured to interface with one of several available layoutEngine **backend libraries**[TODO: Add to References]. The primary library acts as the interface for R users while the backend library is the interface between R and the web browser. The solution design is separated into these components as much of the complexity exists in the backend library. This abstracts much of the complexity away from the user and allows improvements to be made to backend library with little impact to the user.

The primary library is a relatively simple and robust interface. It provides several helper functions to pass graphics-based data between the R user and layoutEngine backend. [TODO: extend description]

The backend library has many complexities to manage which present the primary challenge of this solution. The key mechanisms and complexities the backend library must contend with include:

- Variability in Host Machine
  - Cross-platform system calls (macOS, Windows and Linux)
  - System and R dependencies
- Functionality
  - Locate and manage a modern web browser session
  - Send and receive data between a R session and web browser
  - Query and modify the web-page DOM [TODO: Add glossary term]

There are three layoutEngine backends available for use with the layoutEngine at the time of this research. These backends successfully demonstrate the viability of this approach.

3.4 Benefits

3.4.1 General

- 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

- 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 generaate HTML
  - Knitr (markdown?)
  - xtable
  - htmltools (RStudio `https://www.stat.auckland.ac.nz/~paul/Reports/HTML/layoutengine/layoutengine.html#pkg:htmltools`)
  - Others?
- layoutEngine Backends

3.4.2 DOM Backend

- [TODO]
- Based on Paul's DOM package
- Live visual feedback for debugging, reviewing output
- Access to latest web browser and therefore latest HTML, CSS and JS specs

3.4.3 PhantomJS Backend

- [TODO]
- Simple, lightweight, few dependencies
- Per Ref
  - Based on WebKit browser engine (Apple from Chrome)
  - Does not require a GUI and performs layout off-screen

3.4.4 CSSBox Backend

- Per Ref
  - Based on CSSBox Java library
  - Generates HTML layout information directly (i.e. standalone HTML layout engine)
  - Generates information for every line of text after layout which is better than most web browsers
- [TODO: get some feedback on this from Paul]

3.5 Limitations

3.5.1 General

- Have to learn and write in HTML/CSS/JS
- Security layer around using a browser
- System dependencies across all OS is trouble
- Font managing software customized per OS
- Per Ref
  - Cairo-based R graphics devices (and pdf & postscript)

- ▷ Matching or converting X11 fonts for the X11 device to fonts the layout backends can use would be hard
- ▷ Support for native Windows and MacOS graphics devices
- ▷ Smallish list of CSS that is support in layoutEngine currently do nothing when pushed back to R from the backend
- ▷ Issues with hyphens in CSS (as string variables in R)
- ▷ Pixel resolution compatibility (resolution of graphics device should be set to 96 dpi)
- ▷ Not a fast process (rendering HTMl) => speed cost for expanded functionality

3.5.2 DOM Backend

- ► [TODO]
- ► Default browser opens every call
- ► Per Ref
  - ▷ See article about managing font types

3.5.3 PhantomJS Backend

- ► [TODO]
- ► Lack of visual feedback
- ► No longer developed so will eventually lose support
- ► Based on older WebKit engine so behind on HTML and CSS specs

3.5.4 CSSBox Backend

- ► Per Ref
  - ▷ Have to keep track of levels of accuracy based on what device HTML will be rendered on
  - ▷ Lags browsers support of web standards (modern CSS specifically)
- ► [TODO]

4 layoutEngine Development

4.1 Objectives

The viability of the layoutEngine approach is still being explored and it is the layoutEngine backend where the majority of the limitations reside. There are several existing back-ends however each has certain limitations that must be rectified before community adoption is possible.

This report introduces two newly developed layoutEngine back-ends' which attempt to address the limitations of the existing designs. One relies on a **Selenium** server hosted within a **Docker container** container. The second is a custom **NodeJS** server also hosted with Docker.

4.2 Requirements

The backend must support a browser compatible communication protocol. Data must be transferred between R and the browser in both directions. The backend must first send the raw HTML based data to the browser. It must

1. Explain improvements are needed with the **backend**
2. What does the **backend** need to achieve?
3. What solution designs were considered?
4. How do these compare to each other?
5. What (and why) were the two **backend** designs chosen?

5 RSelenium Backend

5.1 Solution Design

5.2 Benefits

5.3 Limitations

6 NodeJS Backend

6.1 Solution Design

6.2 Benefits

6.3 Limitations

7 Comparison

1. Summary of solution features, benefits and limitations
2. How do they rank with the existing **backends**?

8 Recommendations

1. Overview of layoutEngine as a solution to generating print quality graphics
2. Do the new backends improve its performance?
3. Where should future development work concentrate?

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 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 **kcull\layoutengine-research**. The container is built from the Ubuntu 18.04 image and has R 3.6.3 and Emacs 27.1 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 also has the primary layoutEngine repositories cloned in the `\opt` directory.

Directory Hierarchy:

```
# Emacs configuration files
/home/user/.emacs.d/
# Github repository for research paper
/project/
# Github repository for layoutEngine
/opt/layoutengine
# Github repository for layoutEngineDOM
/opt/layoutenginedom
# Experimental code for layoutEngineRSelenium
/opt/layoutenginerselenium
# Experimental code for layoutEngineNodeJS
/opt/layoutenginenodejs
```

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.

- GitHub Repository: kcullimore/layoutengine-research
- Docker Image: kcull\layoutengine-research

1 - Download the docker image:

```
$ docker pull kcull/layoutengine-research:latest
```

2 - Create a working directory on the host machine and clone the github repository:

```
$ mkdir /home/$USER/layoutengine-research
$ git clone git@github.com:kcullimore/layoutengine-research.git /home/$USER/layoutengine-research
```

3 - Grant local access to your X server to allow Emacs to run in a local window and the run the docker container (setting is reset upon reboot): **Warning: this exposes your computer. Read more here.**

```
$ xhost +local:
```

4 - Run the docker container:

```
$ docker run --rm -it \
         --network host \
         --privileged=true \
         --env DISPLAY=unix$DISPLAY \
         --volume /tmp/.X11-unix:/tmp/.X11-unix \
         --volume /var/run/docker.sock:/var/run/docker.sock \
         --mount type=bind,source=/home/$USER/layoutegine-research/,target=/project/ \
         --name layoutengine-research \
         kcull/layoutengine-research: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 $\{\backslash xhost + local : command)$.

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

- Opens Treemacs with `M-0`
- Open folder structure to `/project/paper/` with Tab-Enter or Mouse
- Open org-mode markdown file `/layoutengine-research-paper.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.1.3 Emacs within Docker Container

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 TEX file (*.tex) is also produced after transpilation. This TEX file can be edited and used with a standard LATEX command: `latex report.tex`.

To be continued...

A.2 Org-mode examples

A.2.1 Font definitions

Using LaTeX 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.2.2 Sample R code highlighting

```r
##*******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.2.3 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.2.4 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.2.5 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

Murrell, P. (2018). “Rendering HTML Content in R Graphics” Technical Report 2018-13, Department of Statistics, The University of Auckland.

References

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

[2] docker socket solution https://jpetazzo.github.io/2015/09/03/do-not-use-docker-in-docker-for-ci/