Improvements to the 'dvir' package

Alexander van der Voorn

Executive summary

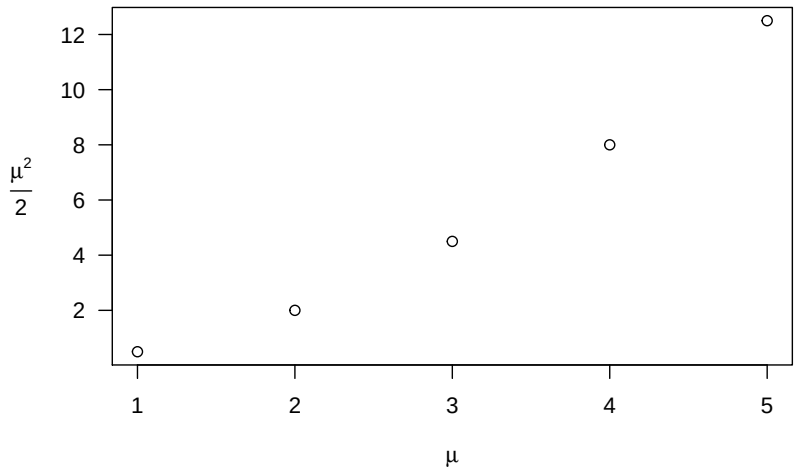
. . . goes here. . . .

Is this the same as an abstract, or something different? Google says they are different (Executive summary = this report condensed to a paragraph, abstract = description of the things you'll find in the report, without explicitly mentioning the actual content)

Introduction

R has the ability to display mathematical symbols and equations in graphics using the "*plotmath*" feature, interpreting everything within a call to `expression()` as a mathematical equation.

```
mu <- 1:5
opar <- par(mar = par()$mar + c(0, 1, 0, 0))
plot(mu, mu ^ 2 / 2, xlab = expression(mu), ylab = "", yaxt = "n")
axis(2, las = 1)
mtext(expression(frac(mu ^ 2, 2)), side = 2, line = 3, las = 1)
```



```
par(opar)
```

This provides us with most of the symbols used for equations, such as brackets and fractions, and formats them in a layout resembling TEX, but it is limited in its fonts. Compare the y-axis label above with how it would look if created in a LATEX document.

$$\frac{\mu^2}{2}$$

Figure 1: The y-axis label from above, if it were created in LATEX

The difference is stark and there are several approaches in R which can get us closer to the LATEX result (Murrell, 2018 [Revisiting Mathematical Equations in R: The 'dvir' package]).

- [**Example:** use `extrafont` and `fontcm` packages, embed CM fonts in PDF]
- [**Example:** use `tikzDevice` package, which creates PGF/TikZ version of plot (and as such converts all text in plot to LaTeX (including labels))]

What we want is a middle ground - being able to harness the power of TEX and its typsetting capabilities in R graphics on our choice of text or equation. This is where the `dvir` package comes in.

The `dvir` package provides a simple user interface, in the style of the R `grid` graphics package [Reference R grid graphics here], by way of the `grid.latex()` function:

- [**Example:** example of grid-based plot, changing labels and/or title with `grid.latex()`. Maybe ggplot2?]

So why the project?

The `dvir` package already worked really well in a lot of cases. There were however plenty more desirable features of TEX and its extensions though that could not yet be implemented in R by `dvir`. The power of this package is from ensuring it is comprehensive enough to meet a user's entire TEX needs in R graphics without having to leave R to do annotations in LATEX itself (or Photoshop/Illustrator!).

By keeping things "in R" users only need to learn R (and basic TEX) code to create their graphics and their work is in one place and easily reproducible. Obviously it may not be realistic to *completely* replicate TEX in R, however there were several aspects of the package identified as having a lot of potential to greatly increase its usefulness. The aspects identified were:

- the speed of the package - anecdotally some complex examples with many `grid.latex()` calls took a while to generate
- expand the capability of creating TikZ drawings by adding support for linear gradient fills
- adding the ability to align TeX text to a *baseline* - the natural line on which characters sit

Background to TeX and `dvir`

TeX

TEX is a program to format and typeset text, and includes some basic macros to do this. LATEX is a higher-level implementation of TEX, essentially consisting of a lot more macros, creating a much more user-friendly interface to TEX. For example, LATEXallows one to create a document with numbered sections, title pages and bibliographies without having to write complicated TEX macros themself. There are other programs that do similar things to LATEX too.

DVI

A TEX or LATEX file is just plain text, so there obviously needs to be a step from

Explain DVI more here (and less in next section explaining dvir packages)

The (pre-existing) `dvir` package

(not sure how to make clear this is all stuff that was in 'dvir' before any of my work (and the fact I didn't change any of this high level stuff))

`dvir`, in a simplified form, works by undertaking the following steps:

- A high level function provided by the package, `grid.latex()`, is called with the TEX code of the expression or text to be displayed on the R graphic.

```
library(dvir)
grid.latex("$x - \\mu$")
```

1. `dvir` creates a TeX document with the expression and a changeable default preamble and postamble.

- [**Example:** TeX document with pre- and post-amble]

2. This TeX document is then processed using the local TEX installation to create a DVI (DeVice Independent) file. A DVI file describes the visual layout of the document to produce but is not specific to any type of output file (PDF, HTML, etc.)
3. The DVI file is read into R. DVI files are binary so are not easily readable by humans but `dvir::readDVI()` allows us to inspect the information inside.

- [**Example:** Extract of DVI file using `readDVI()` (not the whole thing, just the bit relevant to our example $(x - \mu)$)]

4. Three 'sweeps' of the DVI file are completed to extract necessary information about what to display in R (and how to display it):

- Font sweep: Gather the names of all fonts used in the DVI file and locate the relevant font files on the local machine. The font information is stored in a R list as well as a `fontconfig` file.

- Metric sweep: To determine the overall bounding box (size) of the expression to display. This bounding box is used to create a grid viewport which can encompass all of the expression using the native DVI coordinates.
- Grid sweep: Convert all text and symbols into *grobs* (grid graphical objects)

5. These *grobs* are then displayed in the R graphics device.

Code speed (part 1) - remove redundant `font` sweeps

Now that the package "works" there is opportunity to expand its scope and usefulness. In the introduction of this report the case for the `dvir` package was motivated with a simple example of a mathematical equation. `dvir` can be used in larger cases too.

- [**Example:** Thomas Yee's example]

One of the first things investigated in the package was the speed of running the code. Anecdotally, generating any R graphic with non-trivial TEX, like the example above, took a long time and as such improving the time efficiency of the package was very desirous.

To tackle this, and indeed find out whether even was the case, the first task was to profile the existing code to see where the time was being spent whenever the code was run. We used our two examples: the simple $x - \mu$ and our larger example above.

- [**Example:** `profvis()` screenshot of simple example, showing `definePDFFont()` (before change)]
- [**Example:** `profvis()` screenshot of Thomas Yee's example, showing `definePDFFont()` (before change)]
- [**Example:** `profmem()` of simple example, calculate average time taken (before change)]
- [**Example:** `profmem()` of Thomas Yee's example, calculate average time taken (before change)]

Using `profvis::profvis()` we were able to visually explore how the time was spent in these examples. In ten runs of the motivating example, the total average execution time was about 6269ms. Of this, about 3393ms on average, or 54%, was spent in calls to the `dvir` function `definePDFFont()`.

The purpose of `definePDFFont()` is to do a sweep of the DVI file from `grid.latex()` looking for all fonts required, before recording the font names in the font config file, searching the relevant directories for the font files and encoding the fonts. A variable `fonts` is saved with all this information. The further sweeps over the DVI file to determine the bounding box of each character and thus entire image, and to create grid grobs and viewports each redundantly called `definePDFFont()` rather than referring to the already existing variable `fonts` from the first sweep.

This was therefore an easy and quick win - simply changing the subsequent sweeps to ignore the font-defining op code and instead referring to the `fonts` variable created from the font sweep.

- [**Example:** `profvis()` screenshot of simple example, showing `definePDFFont()` (after change)]
- [**Example:** `profvis()` screenshot of Thomas Yee's example, showing `definePDFFont()` (after change)]
- [**Example:** `profmem()` of simple example, calculate average time taken (after change)]

- [**Example:** `profmem()` of Thomas Yee's example, calculate average time taken (after change)]

This resulted in fantastic savings. In 10 runs of the example after this change was made, the average total execution time was 3712ms. This is a reduction of 2557ms, or 41%. The average time spent in `definePDFFont()` was just 1292ms, 62% less than before. A saving of nearly two thirds in the function is consistent with removing two of the three font sweeps based on the code profiling results above.

Code speed (part 2) - font caching

The earlier code speed up was done by stopping `dvir` doing something silly (completing a font sweep three times instead of only one). Our further profiling lead us to find where next our code spends its time and now it was a matter of making `dvir` "smarter".

- [**Example:** `profvis()` result showing `fontEnc()` (I think) taking long time]

- Looks like if we could save/cache a font we could reduce the amount of time to run `grid.latex()`
- `fonts` R list is re-initialised after every call to `grid.latex()`
- Is a font definition in DVI the same over different calls to `grid.latex()`? Yes! even the font def number (a number seemingly determined by TeX)

- [**Example:** font definitions from DVI file (over multiple `grid.latex()` calls) showing same fonts have same def]

- first of all we want the `fonts` R list to persist over multiple `grid.latex()` calls in an R session. We did this by storing fonts list in the `dvir` environment (using `dvir::set()` and `dvir::get()`)
- When come across a font definition (during a font sweep), we check if that font exists - the position in `fonts` list is determined by the font def number, and so as the same fonts (theoretically) have the same font def number, we can compare the new font we've come across with what is existing in that position in the `fonts` list. If nothing exists in that position in the list, then we save it as normal. If a font does exist, then we need to check if it's the same (just in case the font def number is not unique for different fonts across different `grid.latex()` calls)
- To do this easiest way was to expand the stored information about fonts in `fonts` to include the hex code chunk (from DVI) of the font definition
- Then we check if all parts of the new hex code chunk are the same as the existing
- [**Example:** code for new function for checking if two font definitions are the same]
- If the definitions are the same, do nothing. If they are different, overwrite the existing font info with the new font info. This actually removes any concern about using only the font def number (which we're pretty sure stays the same for the same font, but maybe it doesn't)
- Only requirement is that the font def number is unique within a single call to grid.latex() (or rather the resulting DVI output)
- Now only need to change the initialisation (reset) of fonts list to happen on package load, rather than during `grid.latex()` call (because doing it every `grid.latex()` call defeats the purpose of storing fonts).

Occasionally one might still want to reset the font cache, so added an option `options(dvir.initFonts = FALSE)` and added `initFonts = getOption("dvir.initFonts")` to `dviGrob.character()` and `dviGrob.DVI()`

- [**Example:** show function calls with the above, and anything else that helps explain them]

But why to each of these steps? Need to flesh out more why they achieve what we want it to achieve (and any considerations we had in our thought process)

- [**Example:** Profiling results (`profvis()` and `profmem()` showing speed improvement)]

Linear gradient fills

TikZ and `dvir`

TikZ is a TEX package that allows drawing of pictures and diagrams in TEX documents [reference TikZ report/description]:

- [**Example:** simple TikZ drawing (circles with labels, and an arrow maybe)]
- [**Example:** more complicated TikZ drawing, maybe with colouring and stuff]

The original DVI specification only needed to account for text and typesetting (and can do the most basic of rectangles too!), and so was not designed with drawing and graphics in mind. The type of instruction in the DVI file are labelled with an "op code". Each op code described a type of instruction like defining fonts, setting characters to display and vertical and horizontal cursor movements. There were four op codes however, called *DVI specials*, that can contain almost any form of instruction or values needed, such as text colour, to create a document based on the DVI file, such as Postscript or PDF.

The TikZ package uses these DVI specials to describe shapes, drawings and colours in PGF (portable graphics format) which can be translated to instructions for other viewing formats, like Postscript, PDF or SVG. How the instructions are translated is controlled by a TikZ driver. The `dvir` package includes its own TikZ driver to translate the drawing instructions into a form useful to draw the things with R grid graphics [reference Paul dvir TikZ report].

Some TikZ features were not implemented though, notably the ability to have fill colours of shapes as linear or radial gradients or patterns. The primary reason for this is that R did not support these types of fills but the latest R release in May 2021, version 4.1.0, provides support for these fills in the `grid` package, on which `dvir` is built.

- [**Example:** replicate one of the above examples in R]
- [**Example:** TikZ radial gradient fill example]
- [**Example:** Make same TikZ example as above in R with dvir (obviously fill will be blank)]
- [**Example:** Use R 4.1.0 to make a linear gradient in a shape]

As it is, the TikZ driver simply ignores any gradient or pattern fill information when creating the DVI file for `dvir`.

- [**Example:** Use `grid.tikzpicture()` for picture with gradient fill in text, but resulting R graphic does not have fill]

Implementing TikZ gradient and pattern fills in `dvir`

The following steps are required to implement these TikZ fills in `dvir`:

1. Add the fill information (like gradient start and end colours, gradient radius etc.) to the DVI file created by `dvir`
2. Store this fill information during a parse by `dvir` to read the DVI file
3. Add the fill information when drawing the shape in R

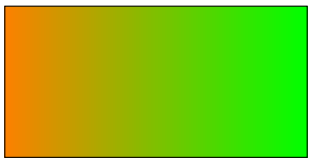
To tackle step 1, we need to update the `dvir` TikZ driver file to include information about the gradient and pattern fills. As the `dvir` TikZ driver file is based on the SVG TikZ driver file, the SVG support for TikZ fills was used as a base to edit to make it specific to `dvir`.

The information we require for the gradient fills from TikZ via the DVI file is as per the arguments for the `grid::linearGradient(...)`, which is used as an argument to `grid::gpar(fill = linearGradient(...))`, which itself is an argument to a `grid` drawing function, for example `grid::grid.rect(..., gp = gpar(fill = linearGradient(...)))`. The most important parts of defining a linear gradient fill is the colours and stops of the gradient fill. The stops of a gradient fill are the locations along the length of a gradient fill where the specified colours are. In between the stops, the gradient between stop colours either side occurs.

The `colours` and `stops` arguments of `linearGradient()` are simply vectors of colours (a character vector of colour names of hexadecimal RGB values) and locations of those colours as a proportion of the distance between the start and end points of the gradient respectively. This obviously guides us as to what information we need to get from TikZ in the DVI file so we can pass it to `dvir`.

Let us consider a simple example, a rectangle with an orange to green linear gradient fill:

```
# Code from TeX should we want to know how to do this in TeX itself
\documentclass{standalone}
\usepackage{tikz}
\begin{document}
\begin{tikzpicture}
\filldraw [draw=black, left color=orange, right color=green] (0,0) rectangle (4,2);
\end{tikzpicture}
\end{document}
```



The following is an extract of the DVI file when the rectangle above is generated using the SVG DVI driver included with the common TEX distributions, `pgfsys-dvisvgm.def`. It has been edited slightly for

readability.

```
xxx1        k=67
            x=dvisvgm:raw <g transform="matrix(1,0,0,1,56.90549,28.45274)">{?nl}
xxx1        k=67
            x=dvisvgm:raw <g transform="matrix(2.26802,0,0,1.134,0.0,0.0)">{?nl}
xxx1        k=66
            x=dvisvgm:raw <g transform="matrix(0.0,1.0,-1.0,0.0,0.0,0.0)">{?nl}
xxx4        k=425
            x=dvisvgm:raw  <linearGradient id="pgfsh2" gradientTransform="rotate(90)">{?nl}
                           <stop offset=" 0.0" stop-color=" rgb(0.0%,100.0%,0.0%) "/>{?nl}
                           <stop offset=" 0.25" stop-color=" rgb(0.0%,100.0%,0.0%) "/>{?nl}
                           <stop offset=" 0.5" stop-color=" rgb(50.0%,75.0%,0.0%) "/>{?nl}
                           <stop offset=" 0.75" stop-color=" rgb(100.0%,50.0%,0.0%) "/>{?nl}
                           <stop offset=" 1.0" stop-color=" rgb(100.0%,50.0%,0.0%) "/>{?nl}
                           </linearGradient>{?nl}
xxx1        k=57
            x=dvisvgm:raw <g transform="translate(-50.1875,-50.1875)">
xxx1        k=97
            x=dvisvgm:raw <rect width="100.375" height="100.375" style="fill:url(#pgfsh2);
                           stroke:none"/>{?nl}
```

We can see from this that the linear gradient definition with stops and colours is defined within a `<linearGradient>` element and given an `id` attribute. In the `<rect>` element a CSS style definition sets the fill of the rectangle by referring to the `id` of the previously defined definition. We can see in the linear gradient definition there are colours defined as RGB values and their respective stops so now we need to get the `dvir` driver file to extract the same information in a "R-friendly" form.

- Go to `dvir` driver file and see what I've added (and try work out what bit does what to explain it! Like it gets defined in one place, then passed to another function to do something else)
- See my meeting progress notes on the matter
- What have we had to change in driver file (and why?) - like specific bits of driver file
- Before and after of DVI file for linear gradient fill (see new information displayed)
- Why couldn't we go further?
- Next steps

Text baselines

- Demonstrate problem (with example using grid.text), especially try multi line text maybe?
- Describe algorithms for determining baselines one by one. In dviMoves, describe then the issues with choosing which one, and the potential algorithms for that
- Describe function I made to calculate all baselines using these methods

- Show result of all this :) (in LaTeX)

Conclusion/summary/Next steps

Bibliography

References to maybe include (if/where relevant):

- All 'dvir' tech reports
- Donald Knuth for TeX
- PGF/TikZ
- R intro
- R base graphics?
- R grid graphics