Improvements to the ‘dvir’ Package

Alexander van der Voorn

Contents

1 Executive summary

. . . goes here. . .

2 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)
```

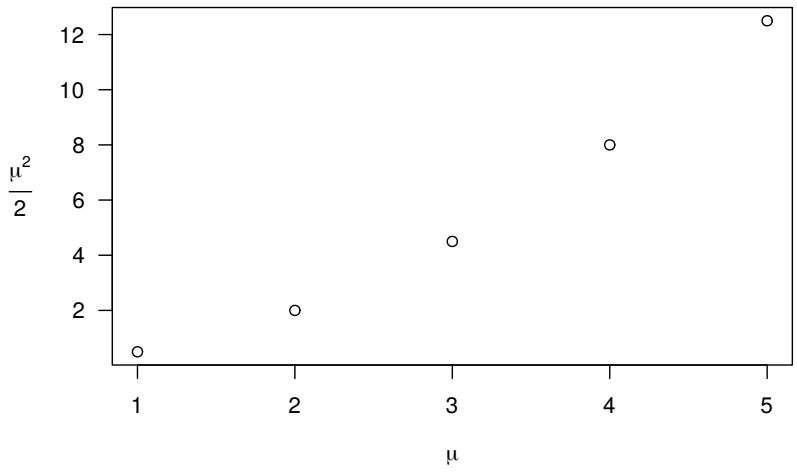


Figure 1: A plot with axis labels made using 'expression()'.

```
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 looks when created by LATEX in figure **??**.

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

Figure 2: 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 on our choice of text or equation in R graphics. This is where the `dvir` package comes in - providing 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?]

2.1 Where this project fits in

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 had not yet been implemented 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 it took a while to generate graphics, especially if there were many `grid.latex()` calls
- expand `dvir`'s capability of creating TikZ drawings by adding support for linear gradient fills
- adding the ability to align text from `grid.latex()` to a baseline - the natural line on which characters sit

3 Background

3.1 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, basically consisting of a lot more macros, creating a much more user-friendly interface to TEX. For example, LATEX allows one to create a document with numbered sections, title pages and bibliographies without having to write complicated TEX macros themself. There are other extensions to TEX that do similar things to LATEX too.

3.2 DVI

A TEX or LATEX file is just plain text, so there needs to be a step to translate this plain text to what you will see on a formatted document on a screen or page. A DVI (DeVice Independent) file is a binary file *describing* the layout of the document. For example, the height of the page, what characters to display and where, and the fonts to be used.

3.3 The (pre-existing) `dvir` package

In a simplified form, `dvir` works by providing a high level function, `grid.latex()`, to call with the TEX code of the expression or text to be displayed.

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

Figure 3: Using ‘dvir‘ to make our caption label from earlier

The following steps are taken when `grid.latex()` runs:

1. A TEX document is created 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.

3. The DVI file is read into R. As DVI files are binary they are not easily readable by humans but the `dvir` function `readDVI()` translates the DVI file into readable text.

- [**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 where 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 the entire TEX passed to `grid.latex()` 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 as per the `grid` package.

4 Code speed (part 1) - removing redundant font sweeps

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 on a larger scale too.

```
xpos <- c(0, 0.25,  0.7, 1)
myplot <- function(abcd = "(a)", col = "black") {
  plot(1:9, 1:9, type = "n", xlim = c(0, 1), ylim = c(0, 1),
       bty = "n", axes = F, xlab = "", ylab = "")
  arrows(0.5, 1, xpos, 0, length = 0.12, lwd = .7,
         col = col)  # All the arrows
  text(0.05, y = 1.1, xpd = TRUE, labels = abcd, cex = 1.0,
       font = 1, col = col)
}  # myplot
par(mfrow=c(2,2),
    mar = c(2.6, 4, 1.5, 2) + 0.1,
    font = 3,  # italic
    las = 1)
myplot()
## Convert to grid
library(gridGraphics)
grid.echo()
## Make arrows "nicer" ?
grid.edit("arrows", grep=TRUE,
          arrow=arrow(angle=10, length=unit(.12, "in"), type="closed"),
          gp=gpar(fill="black"))
## Navigate to plot window
downViewport("graphics-window-1-1")
## Use 'dvir' to draw labels
grid.latex("\\dots", x = 0.44, y = -0.1, default.units="native")
grid.latex("$Y_* =$",
           x = 0.5, y = 1.1, default.units="native")
grid.latex("$a_1$", xpos[1], y = -0.1, default.units="native")
grid.latex("$a_2$", xpos[2], y = -0.1, default.units="native")
grid.latex("$a_{L_A}$", xpos[3], y = -0.1, default.units="native")
grid.latex("$Y_{\\pi} | Y_{\\pi} \\notin \\cal{A}$",
           x = xpos[4], y = -0.1, default.units="native")
grid.latex("$\\omega_1$",
           x = 0.18, y = 0.50, default.units="native")
grid.latex("$\\omega_2$",
           x = 0.32, y = 0.50, default.units="native")
grid.latex("$\\dots$",
           x = 0.44, y = 0.50, default.units="native")
grid.latex("$\\omega_{L_A}$",
```

```
          x = 0.54, y = 0.50, default.units="native")
grid.latex("$1 - \\sum_{s=1}^{L_A}\\omega_s$",
          x = 0.95, y = 0.50, default.units="native")
```

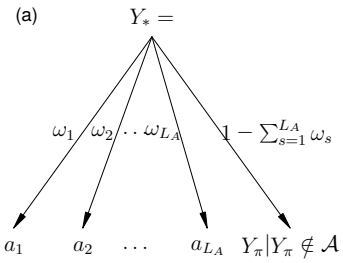


Figure 4: A more complicated example using ‘dvir‘

The example in figure @ref{fig:yeeExample} uses nine calls to `grid.latex()` and was created by a University of Auckland lecturer using the `dvir` package to help write an assignment.

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 that in figure 4, took a long time so it was desirable to see if we could speed it up.

To look into this the first task was to profile the existing code to let us see where in the package time was being spent. This was in `dvir` version 0.2-1.

We visualised the profiling results using `profvis::profvis()`.

We can see the function call stack in figure @ref(fig:profilingSimpleProfvis_0.2-1). At the bottom is the call to `grid.latex()`, which immediately calls `grid.draw()` which in turn calls `latexGrob()`. This calls `readDVI()` for about the first 20ms, then `dviGrob()` for the remaining time to the end of the original `grid.latex()` function call, and so on up the function call stack.

The `profvis::profvis()` output for our more complicated example, in figure @ref(fig:profilingYeeProfvis_0.2-1_highlight) reveals most of the time to create the figure is in `grid.latex()`. Note that the code to draw the arrows and the “(a)” in this example is so quick it occupies the very skinny call stack on the far left of the graph. `grid.latex()` and its subsequent function calls, on the other hand, take up most of the time required to produce the example.

In figure @ref(fig:profilingYeeProfvis_0.2-1_highlight) some blocks in the call stack have been highlighted - these are related to the `engine$fontDef` operation occuring. This is a part of the “font sweep”, as was

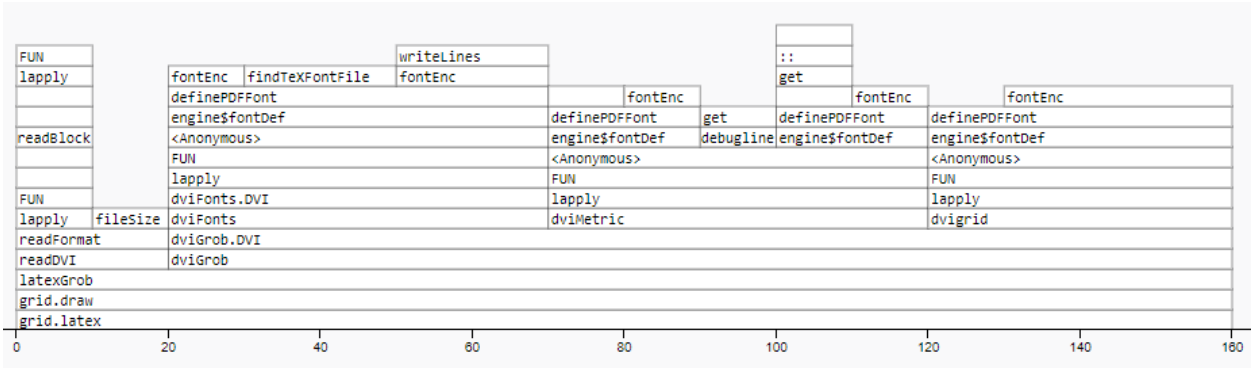


Figure 5: Screenshot of `profvis::profvis()` output for the code `grid.latex("$x - \\mu$")` in `dvir` version 0.2-1.

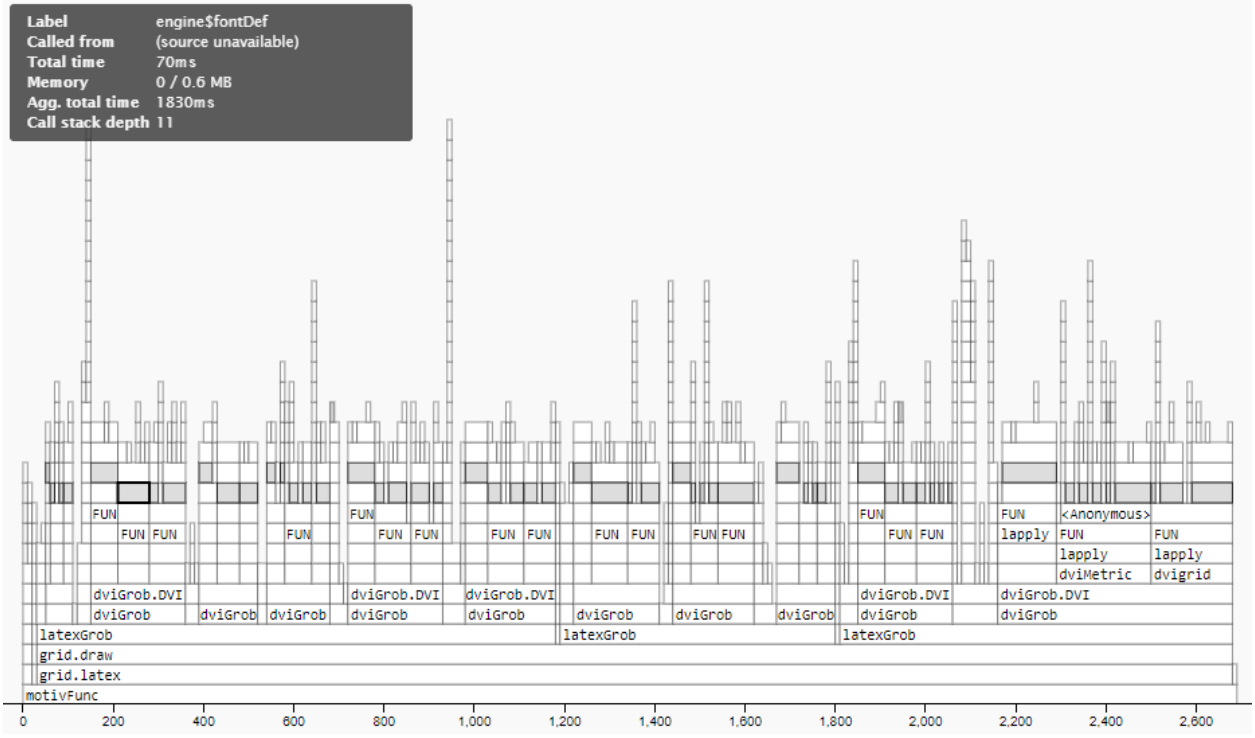


Figure 6: Screenshot of `profvis::profvis()` output for the code creating figure 4 highlighting the time spent in `engine$fontDef`.

described in the introduction to `dvir` in section 3.3.

In the top left corner of figure @ref(fig:profilingYeeProfvis_0.2-1_highlight) we are told the aggregate time spent with `engine$fontDef` is 1830ms. Compared to the total time of this run (a total of about 2700ms), `dvir` is spending a *lot* of time doing these font sweeps.

What was interesting though was that after the actual font sweep the following sweeps for the metric and grid information *also* called `engine$fontDef`. As the point of the font sweep is that it finds all the font information to be used later on the following metric and grid sweeps should not need to "re-sweep" for the fonts.

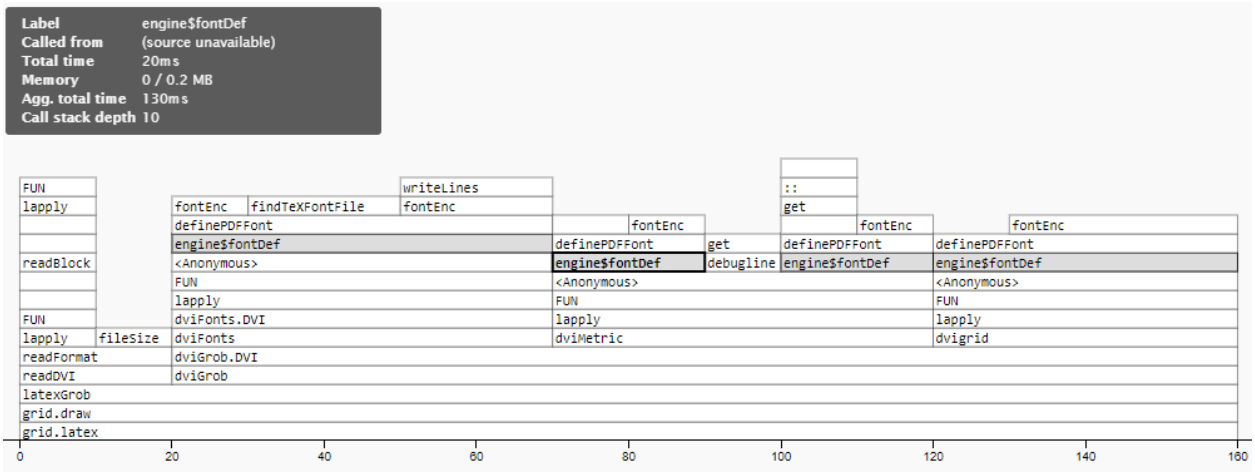


Figure 7: Screenshot of `profvis::profvis()` output for the code `grid.latex("$x - \\mu$")` in `dvir` version 0.2-1, highlighting `engine$fontDef`.

The effect of this is very obvious in figure @ref(fig:profilingSimpleProfvis_0.2-1_highlight) which is the same as figure @ref(fig:profilingSimpleProfvis_0.2-1) but highlights the time spent in `engine$fontDef`. The wrappers for the font, metric and grid sweeps are `dviFonts()`, `dviMetric()` and `dvigrid()` respectively (sixth call from the bottom of the stack). Here we can see nearly all of the time spent in the metric and grid sweeps are actually redoing the font sweep!

The change to be made was simply stopping the metric and grid sweeps from doing the font sweep again.

The font sweep looks in the DVI file for op codes 243 to 246. These are the op codes for font definitions and define the name of a font and give it an identifier to reference in the DVI file when it wants to use that font to display a character.

```
metric_info_243 <- op_font_def
```

```
grid_op_243 <- op_font_def
```

Figures @ref{fig:metricFont_0.2-1} and @ref{fig:gridFont_0.2-1} show the code in the `dvir` package itself where the metric and grid sweeps also redid the font sweep. `op_font_def` is a function which takes the font definition in the DVI file related to that instance of the op code and searches for and records the font information.

```
metric_info_243 <- op_ignore
```

```
grid_op_243 <- op_ignore
```

Figures @ref{fig:metricFont_0.2-2} and @ref{fig:gridFont_0.2-2} show the what the code was changed to in `dvir` version 0.2-2. `op_ignore` is an empty function, so when the metric or grid sweeps comes across that op code, they now do nothing.

Unfortunately these changes by themself caused an error when running `grid.latex()`. This is because one task undertaken before the font sweep is to reset or overwrite the global fonts list (which the font sweep then writes to). The metric and grid sweeps were also doing this even though it was only intended for it to be done by the font sweep. This meant after the font sweep was completed it was overwritten by the metric and grid sweeps and so when `dvir` tried to draw the characters there was no font information to refer to.

The resetting of the global fonts list was initiated when the sweeps passed op code 247 in the DVI file, which is the preamble at the start of every DVI file.

```
metric_info_247 <- op_ignore
```

```
grid_op_247 <- op_ignore
```

Setting the metric and grid sweeps to do nothing when they pass the preamble of the DVI file, again by way of `op_ignore`, solved this problem as the global fonts list created by the font sweep is now not overwritten.

To quantify the impact this has on code speed we recorded the time to run our examples 20 times, after an initial run to compile the package after it was loaded. The first table details the total time spent, in seconds, in each of these functions in the 20 runs before and after these changes were made.

The second table contains the change in time as a proportion of the "before" time.

before_gridlatex	before_metric	before_grid	after_gridlatex	after_metric	after_grid
7.72	2.24	1.96	5.39	0.54	0.34
115.98	34.04	31.67	61.96	5.66	3.56

gridlatex_change	metric_change	grid_change
-0.3018135	-0.7589286	-0.8265306
-0.4657700	-0.8337250	-0.8875908

Section to be continued with:

- Formatting above table better (rounding, units, convert to percentages, better headings, centre align etc.)
- Text/paragraph description of these results

4.1 Profiling environment specifications

The exact results obtained are specific to the computing environment used. Specific details are provided below. The sampling nature of profiling (intermittent recording of the call stack) will give different results every time it is done.

The profiling results are very specific to the computer setup used and could change considerably depending on the exact computing environment in which the `dvir` package is used.

The profiling results in this report, in this and the next section, were calculated with the following setup:

- A virtual machine via Oracle VM Virtualbox
- Virtual machine running Ubuntu 18.04.5 LTS
- R version 3.4.4
- `dvir` package versions as described with the profiling results

5 Code speed (part 2) - font caching

The earlier code speed up was done by stopping `dvir` doing something "silly". 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)]

6 Linear gradient fills

6.1 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]

6.2 Implementing TikZ linear gradient 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

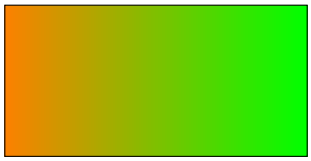
To tackle step 1, we need to update the `dvir` TikZ driver file to include information about the gradient and pattern fills. As the `dvir` TikZ driver file is based on the SVG TikZ driver file, the SVG support for TikZ fills was used as a base to edit to make it specific to `dvir`.

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 or reduce/simplify to print above t
\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.

Section to be continued with:

- What have we had to change in driver file (and why?) - like specific bits of driver file
- Before and after of DVI file (using new driver) for linear gradient fill (see new information display). Do the "after" for both simple rectangle and maybe a more complicated example?
- Why couldn't we go further? Mention the transformations?
- Next steps (to complete steps 1, 2 and 3 as detailed earlier), including discussing how this applies to radial gradient fills and pattern fills

7 Text baselines

- Do all this without examples first maybe, as that will be some fiddly work?
- 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)
- Do I need to explain what the 'down' moves are? And that that's oretty much the cursor movement?

7.1 The problem

Text characters have a baseline, that is, a horizontal line on which the characters naturally sit so all the letters appear to be in line with each other. Some letters, like a lower case p or j, have a "descender". A descender is the part of a character that sits *below* the baseline. `grid.text()` accounts for this baseline so when you bottom align text at a certain y value, the descenders will actually fall below the y value we defined, despite the bottom justification. This makes sense because nearly every piece of written text is like this. Unfortunately, `grid.latex()` doesn't account for text baselines. It will simply do any alignment in relation to bounding box of the text.

To fix this, `dvir` needs to obtain or calculate a value for the baseline for any given piece of text to offset the bounding box when it is drawing text. All of `dvir`'s information comes from the DVI file we either need to get a baseline value from the DVI file itself, or calculate it *from* information in the DVI file. Unfortunately DVI files do not state a baseline value so we will have to explore some possible methods to determine a baseline value. These methods are referred to as algorithms from here on due to their heuristic nature.

7.2 Implementation

To explore the practicality of the algorithms detailed below and evaluate their usefulness an R function, `baselines()` has been created as part of this project. This function takes several arguments, including the desired baseline selection algorithm, any other information needed for that particular algorithm, and the TEX code as you would use with `grid.latex()`. The output of this function is the distance, or in some case distances, from the bottom of the bounding box of the text to the possible baseline value. These distances are return as `grid` units.

Now that the potential baselines have been calculated, if when displaying the text we do not use the specified y value, but rather move the bounding box down by the amount of the baseline height, it will be the baseline of the text that is equal to the y-value.

This function has been written to easily allow integration of other algorithms and most of the function should be able to be directly implemented in the `dvir` package, should this baseline algorithm feature be implemented into `dvir` formally.

7.3 Our potential solutions

We explored several different algorithms to calculate the baseline for several different types of text that could be used with `grid.latex()`. These algorithms are detailed below.

7.3.1 `alex` algorithm

This is a simple algorithm which was determined after inspection of some DVI files. In every DVI file, there is a statement specifying the size of the bounding box of the text.

- **Example** of of the HiResBoundingBox statement here

After this statement there appears to consistently be a downward move the height of the bounding box of text, and then a move upward before the first character is drawn. This algorithms take the cursor location after that upward move to be the baseline. In instances where the entire text has no descenders i.e. the baseline is the bottom of the bounding box, there will be no upward move before the first character. In this case a value of 0 is returned as the baseline.

Here is an example of the `baselineAlgorithms()` function in use...

... and here are some examples showing the text and the calculated baseline from the function...

7.3.2 `dviMoves` algorithm

This is an extension of the `alex` algorithm. Rather than only taking the location after the second 'down' move, this algorithm keeps track of *all* the up and down moves of the cursor. The motivation behind this is that the upward and downward "moves" in the DVI file reflect the cursor moving to the baseline value of the next character to be typeset. Once again we assume that the first downward move after the "HiResBoundingBox" statement id from the top to the bottom of the bounding box and so this algorithm only returns the upward and downward cursor moves from there, however as DVI files have the ability to save the current cursor location, move around a bit, then reset back to the saved location, all up and down moves are recorded from the start of the DVI file.

There are two complications with this method:

- As it returns all the vertical positions the cursor moves to there are many possible "baselines" returned. Any more than one means we have to decide which of the baselines values to actually choose
- There are often several up and downward moves in the DVI file between typeset characters so in between the "useful" baselines there can be some which are not so useful or duplicates

To account for these considerations, along with the `dviMoves` algorithm, the function also allows a choice of method to select a *single* baseline out of the usually many returned by the algorithm. These methods are:

7.3.2.1 `dviMoves` selection method `all`

7.3.2.2 `dviMoves` selection method `index`

7.3.2.3 `dviMoves` selection method `bottomUp`

7.3.2.4 `dviMoves` selection method `nextChars`

7.3.2.5 Other potential `dviMoves` selection methods (not implemented)

- best guess
- prev char
- extend dviMoves to only record position just before a character is typeset?

7.3.3 `preview` algorithm

7.3.4 `dvipng` algorithm

7.3.5 any other algorithm?

7.4 Discussion of algorithms

Overall, the `dviMoves` algorithm has performed best out of all of these. While most of these algorithms perform well for most of the examples, the `dviMoves` algorithm performs well for *all* the examples, notably for giving the option to align the baseline of *any* line of multi-line text. Utilising the fact it returns all baseline values for all displayed characters, it is possible to align, for example, with any character in a mathematical equation, whether it be of a different size or a superscript or subscript.

7.5 Next steps to integrate with `dvir` package

- mention the "matching' algorithm of dviMoves (to compare with another algorithm)
- mention the "just ask TeX for answer" method

```
# source(algorithms.R)
```

8 Conclusion/summary/next steps

. . . goes here. . .