Improvements to the ‘dvir’ Package

Alexander van der Voorn

Bachelor of Science (Honours)
Department of Statistics
The University of Auckland
New Zealand

Contents

1 Executive summary

TeX has many desirable features, such as math equation formatting and typesetting, which are currently not available in R. The `dvir` package extends R to draw TeX output on R graphics. By extending the usability and functionality of the `dvir` package it better streamlines the workflow of making R graphics that include some TeX output. Rather than making some graphics in R and exporting to LaTeX itself or an image editor like Photoshop to add TeX or TeX-like output, this can all be done in R through a simple interface from the `dvir` package. Three improvements to the `dvir` package are investigated and discussed in this report.

Running functions from the `dvir` package took a long time. The speed of the package was increased by making two changes - removing redundant code that meant the `dvir` package was searching for fonts three times instead of once, and allowing font information to be cached so that once font information was found the `dvir` package did not need to search again for that same information.

The latest version of R adds functionality of linear and radial gradient fills. This means gradient fill information from Ti*k*Z graphics, a TeX package, could be generated by `dvir` and manipulated to be fed into R. This was not fully implemented as the complexity of the gradient definition from Ti*k*Z resulted in "warped" gradient fills which are not possible in R. This work could continue once support for these "warped" gradients are added to R.

The `dvir` package currently performs any alignment based on the bounding box of the text but aligning with the baseline of the text is more desirable. TeX output does not explicitly state a baseline value for a piece of text so some algorithms for determining the text baselines were implemented in an R function to test their usefulness. One of these algorithms performs particularly well for all tested examples, including multiline text.

The magnitude of the speed increase alone makes `dvir` much more practical as it allows much faster testing of code and allows for more complicated graphics with many calls to the `dvir` package. The ability to align text to its baseline means graphics with the addition of TeX, for example in lattice plot headers, can be aligned with one another without requiring "touching up" outside of R.

This report, source code and associated files can be found on GitHub[1].

[1] https://github.com/ajvandervoorn/honours_dissertation

2 Introduction

R (R Core Team 2021) has the ability to display mathematical symbols and equations in graphics using the "plotmath" feature (Murrell and Ihaka 2000), interpreting everything within a call to `expression()` as a mathematical equation.

```
x <- 1:5
y <- x ^ 2 / 2
expr <- expression(frac(mu ^ 2, 2))
plot(x, y, xlab = expression(mu), ylab = "", yaxt = "n", type = "l")
axis(2, las = 1)
mtext(expr, side = 2, line = 3, las = 1)
```

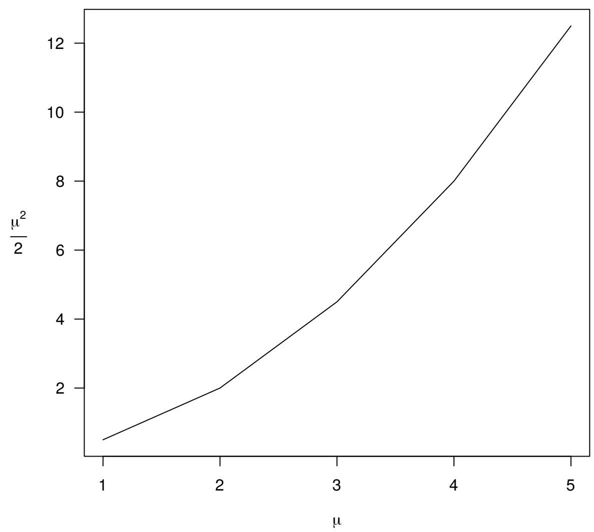

Figure 1: A plot with axis labels made using `expression()`.

This provides us with most of the symbols used for equations, such as brackets and fractions, and formats them in a layout resembling TEX (Knuth 1986), but it is limited in its fonts. Compare the y-axis title above with how it looks when created by LATEX in figure 2.

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

Figure 2: The y-axis title from above, created with LATEX.

The difference is stark and Murrell (2018) describes several approaches, and their limitations, in R which can get us closer to the LATEX result as motivation for the `dvir` package. The approaches are replicated here.

The `extrafont` package (Chang 2014) with the `fontcm` package extension (Chang, Kryukov, and Murrell 2014) works for this particular example but not if we wanted to include other TEX symbols.

```r
library(grid)
library(extrafont)
font_install('fontcm')
loadfonts("pdf")
pdf("extrafont.pdf", width = 1, height = 1)
grid.text(expr, gp = gpar(fontfamily = "CM Roman"))
dev.off()
embed_fonts("extrafont.pdf", outfile = "extrafont-embed.pdf")
```

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

Figure 3: The axis title using the `extrafont` package.

```r
expr2 <- expression(bgroup("(", frac(x - mu, sigma), ")"))
pdf("extrafont-2.pdf", width = 1, height = 1)
grid.text(expr2, gp = gpar(fontfamily = "CM Roman"))
dev.off()
embed_fonts("extrafont-2.pdf", outfile = "extrafont-2-embed.pdf")
```

$$\left(\frac{\mathrm{x} - \mu}{\sigma}\right)$$

Figure 4: Another equation using the `extrafont` package.

In the case of figure 4, note how `extrafont` has once again used the correct font for the greek symbols, but does not quite replicate the rest of the result from LaTeX, as in figure 5.

$$\left(\frac{x - \mu}{\sigma}\right)$$

Figure 5: A LaTeX equation with brackets.

```r
library(tikzDevice)
options(tikzDocumentDeclaration = "\\documentclass[12pt]{article}")
packages <- c(getOption("tikzLatexPackages"),
              "\\usepackage{amsmath}\n")
options("tikzLatexPackages" = packages)

tikz("tikzDevice.tex", standAlone = TRUE)
```

```r
opar <- par(mar = par()$mar + c(0, 1, 0, 0))
plot(x, y, xlab = "$\\mu$", ylab = "", yaxt = "n", type = "l")
axis(2, las = 1)
mtext("$\\dfrac{\\mu^2}{2}$", side = 2, line = 3, las = 1)
dev.off()
```

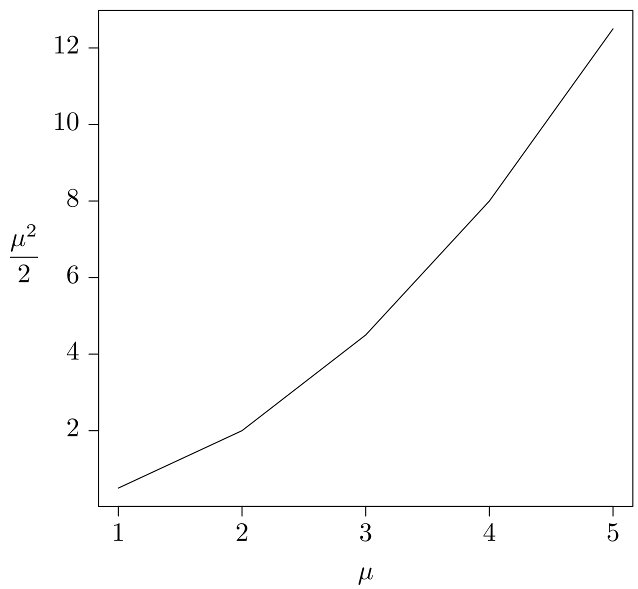

Figure 6: Drawing the plot with the `tikzDevice` package.

The `tikzDevice` package in R will draw the axis titles with TEX but will also use TEX for all other text on the plot, like the axis labels in figure 6. This may be appropriate if the graphic is to be included in a TEX document but not so much for other document formats like HTML.

What we want is a middle ground - being able to harness the power of TEX and its typesetting 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 R `grid` graphics, by way of the `grid.latex()` function. Figure 7 recreates figure 1, but with the axis titles made using `grid.latex()`.

```r
library(dvir)
preamble <- "\\documentclass[12pt]{standalone}\n\\usepackage{amsmath}\n\\begin{document}"
par(mar = par()$mar + c(0, 1, 0, 0))
plot(x, y, xlab = "", ylab = "", yaxt = "n", type = "l")
axis(2, las = 1)

grid.latex("$\\mu$", x = 0.545, y = 0.07)
grid.latex("$\\dfrac{\\mu^2}{2}$", x = 0.06, preamble = preamble)
```

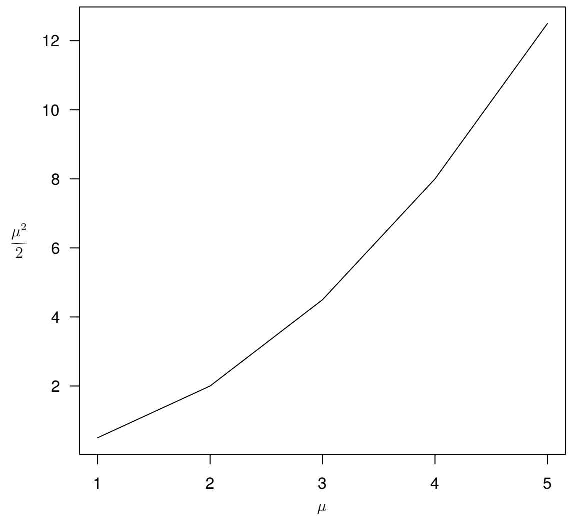

Figure 7: Changing axis titles with `grid.latex()`.

2.1 Where this project fits in

The `dvir` package already worked really well in a lot of cases. There were however a number of desirable features of TEX and its extensions that had not yet been implemented by `dvir`. The power of this package comes 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 other software like Adobe Photoshop or 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. It may not be realistic to *completely* replicate TEX in R, however several aspects of the package were identified as having the potential to 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 Ti*k*Z 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 that sits on top of TeX, essentially consisting of a lot more macros, creating a much more user-friendly interface to TeX. For example, LaTeX allows the creation of a document with numbered sections, title pages and bibliographies without having to explicitly include complicated TeX macros.

3.2 DVI (DeVice Independent)

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 TeX "engine" like `pdflatex` normally processes TeX to PDF but it can also produce a DVI file. A DVI 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. DVI provides a format that can be read into R so that we can draw the TeX layout in R. This is what the `dvir` package does.

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. In this instance, the preamble is changed only because the axis title uses `\dfrac` so the `amsmath` package needs to be specified in the preamble.

```r
library(dvir)
texCode <- "$\\dfrac{\\mu^2}{2}$"
newPreamble <-
  "\\documentclass[12pt]{standalone}\n\\usepackage{amsmath}\n\\begin{document}"
grid.latex(texCode, preamble = newPreamble)
```

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

Figure 8: Using the `dvir` function `grid.latex()`.

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.

   ```
   \documentclass[12pt]{standalone}
   \usepackage{amsmath}
   \begin{document}
   $\dfrac{\mu^2}{2}$
   \end{document}
   ```

2. This TeX document is then processed using the local TeX installation to create a DVI 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. This is an extract of a DVI file after using `readDVI()`, stating the definition of a font and then "setting" (i.e. printing) the character "2".

```
fnt_def_1    fontnum=15, checksum=2088458503, scale=524288,
             design=524288, fontname=cmr8
set_char_50  '2'
```

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. The `dvir` package can be used on a larger scale too.

```
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")
```

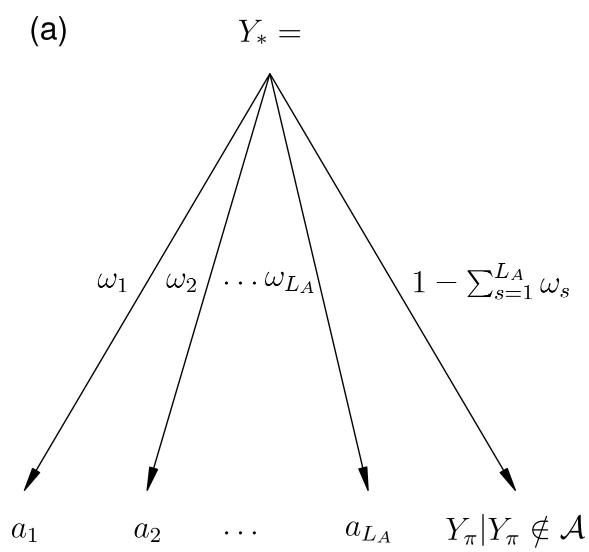

Figure 9: Using nine `grid.latex()` calls with base R graphics.

The example in figure 9 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 9, took a long time so it was desirable to 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. The profiling results were visualised with `profvis::profvis()`.

We can see the function call stack in figure 10. On the fifth line from the bottom is the call to `grid.latex()`, which immediately calls `grid.draw()` which in turn calls `latexGrob()`. This calls `readDVI()` for about the

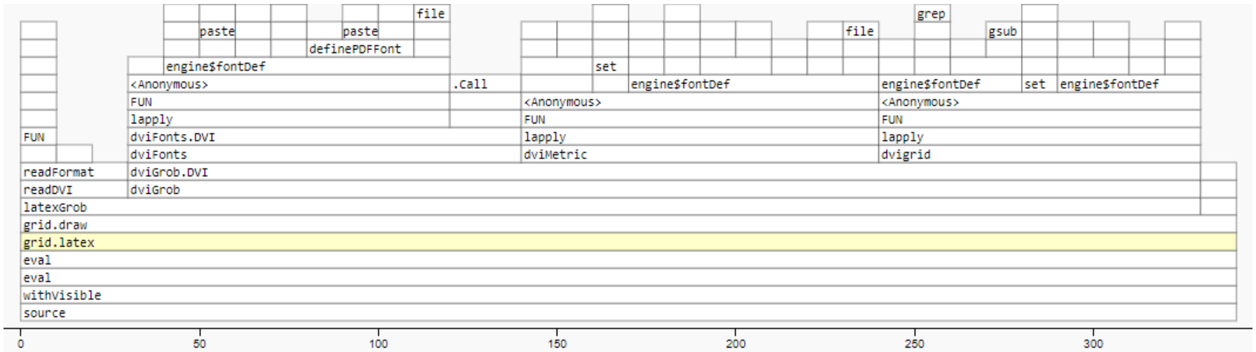

Figure 10: Screenshot of `profvis::profvis()` output for the simple example in `dvir` version 0.2-1.

first 30ms, then `dviGrob()` for the remaining time to the end of the original `grid.latex()` function call, and so on up the function call stack.

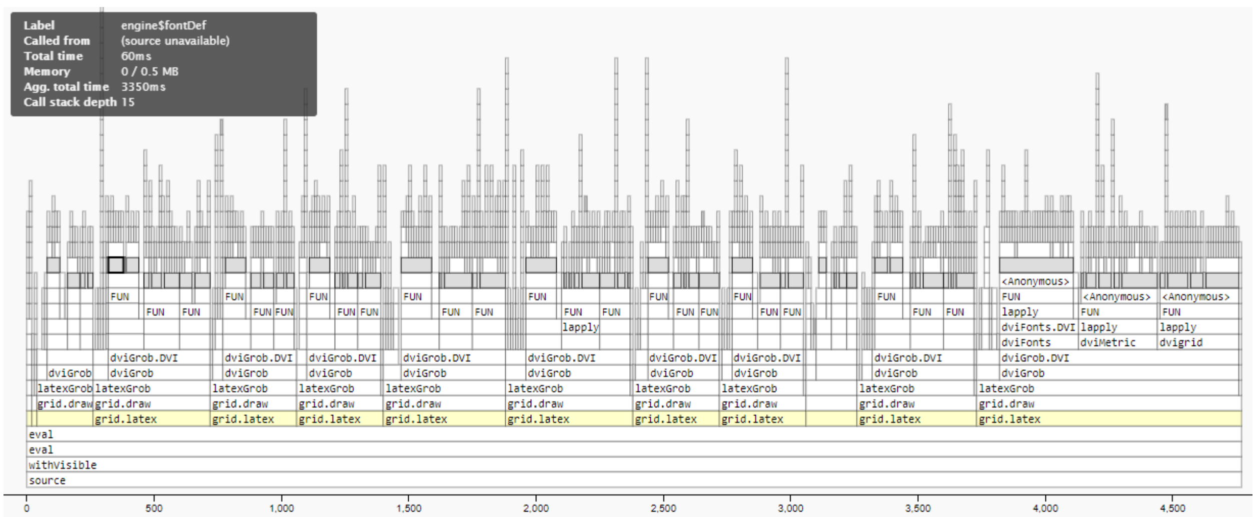

Figure 11: Screenshot of `profvis::profvis()` output for the code creating the example at the start of this section, highlighting the time spent in `engine$fontDef`.

The `profvis::profvis()` output for our more complicated example, in figure 11, reveals most of the time spent creating 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. On the other hand, `grid.latex()` and its subsequent function calls take up most of the time required to produce the example.

In figure 11 some blocks in the call stack have been highlighted light grey. These are related to the `engine$fontDef` operation occurring. This is a part of the "font sweep", as was described in section 3.3.

In the top left corner of figure 11 we are told the aggregate time spent with `engine$fontDef` is 3350ms. Compared to the total time of this run (a total of about 4750ms), `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 intention 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.

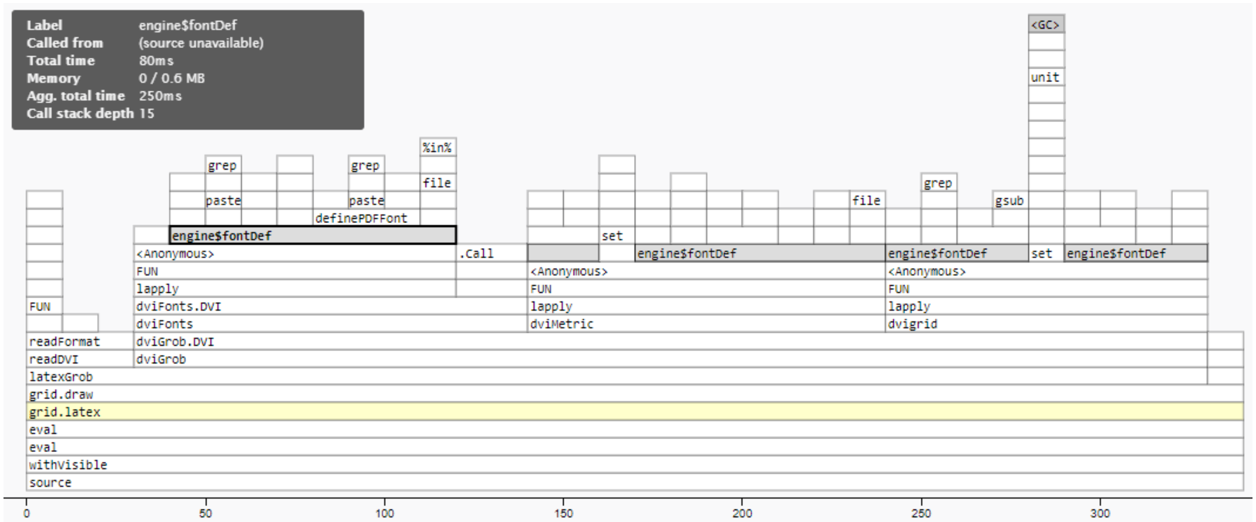

Figure 12: Screenshot of `profvis::profvis()` output for the simple example in `dvir` version 0.2-1, highlighting `engine$fontDef`.

The effect of this is very obvious in figure 12 which is the same as figure 10 but highlights the time spent in `engine$fontDef` in light grey. The wrappers for the font, metric and grid sweeps are `dviFonts()`, `dviMetric()` and `dvigrid()` respectively (tenth call from the bottom of the stack). Here 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.

The following code is taken from the the `dvir` package, showing where the metric and grid sweeps also redid the font sweep. The function `op_font_def` 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_font_def
grid_op_243 <- op_font_def
```

The following code shows what the code was changed to in `dvir` version 0.2-2. The function `op_ignore` is an empty function, so when the metric or grid sweeps comes across that op code, they now do nothing.

```
metric_info_243 <- op_ignore
grid_op_243 <- op_ignore
```

Unfortunately these changes alone 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. 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.

```
metric_info_247 <- op_ignore
grid_op_247 <- op_ignore
```

To quantify the impact this has on code speed the time to run the examples 20 times was recorded, after an initial run to compile the package after it was loaded. The average of these 20 runs was then calculated.

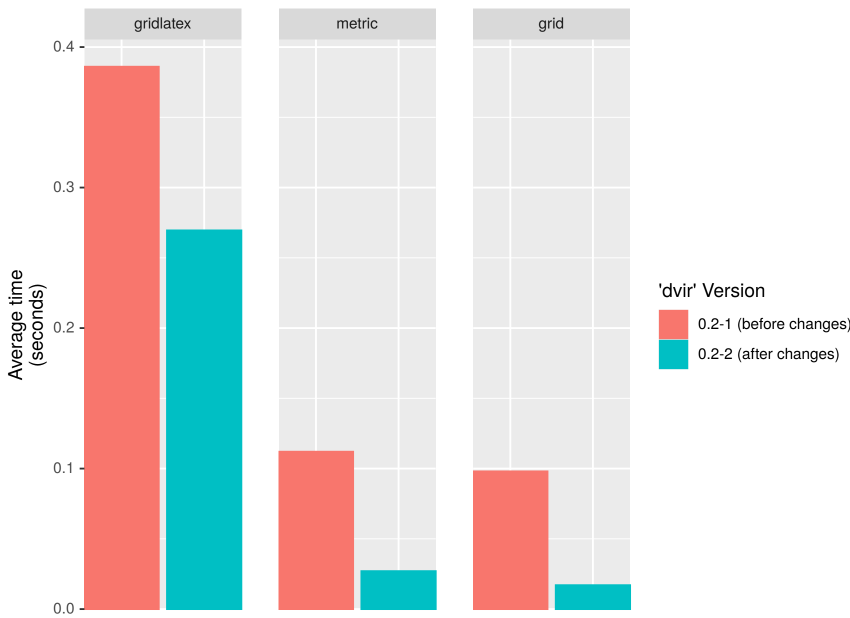

Figure 13: The average time spent in the `grid.latex()` function, metric sweep and grid sweep before and after these changes, over 20 runs of the simple example.

For the simple example the time spent in the metric and grid sweeps have decreased by 76% and 83% respectively. The speed of the overall `grid.latex()` call has decreased by 30%.

Similarly for the more complicated example in figure 14 the metric and grid sweeps have decreased by 83% and 89% respectively, with `grid.latex()` overall taking 47% less time.

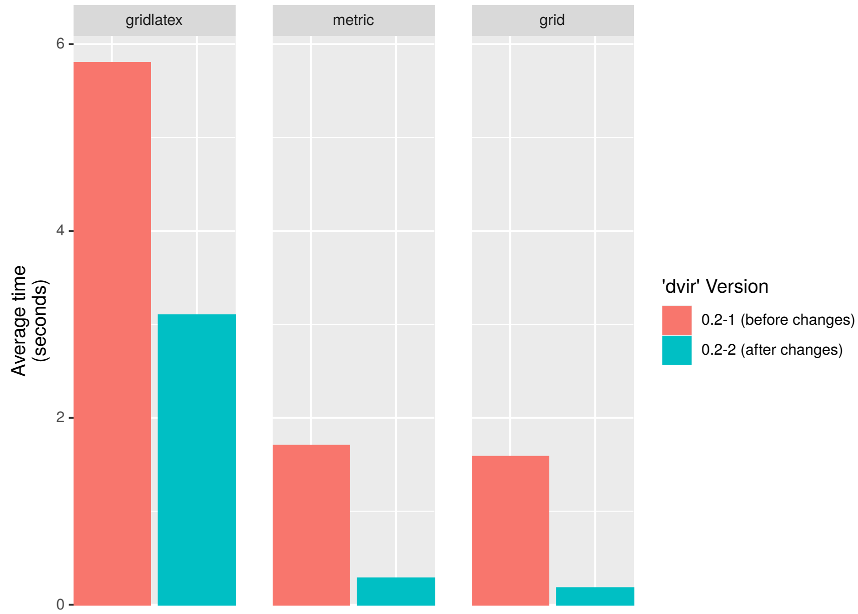

Figure 14: The average time spent in the `grid.latex()` function, metric sweep and grid sweep before and after these changes, over 20 runs of the complicated example.

5 Code speed (part 2) - font caching

In the previous section, the speed of `dvir` was increased by stopping it doing something “silly”. The profiling results from the previous section demonstrate how much time `grid.latex()` spends dealing with fonts so this section is about making `dvir` “smarter” by caching the information it gathers about fonts between `grid.latex()` calls.

For example, creating figure 9 requires three `grid.latex()` calls for the same symbol ω. Each time, the following font definition appears in the DVI file for `grid.latex()` to search for during its font sweep.

```
fnt_def_1    fontnum=17, checksum=-1209964637, scale=786432, design=786432,
             fontname=cmmi12
```

In fact, that example’s 11 `grid.latex()` calls define a font 29 times in total, for eight unique fonts.

Caching fonts requires several steps:

- During the font sweep, search for all font definitions as per usual.
- For each font definition, check if the font has been cached already.
  - If yes, do nothing. Refer to the existing font information in the cache when the font needs to be used.
  - If no, search for and store the system font information as per usual.

The font information is stored in R as a list. Every time `grid.latex()` processed a “preamble” op code in a DVI file, that is, every time `grid.latex()` was called, the font list was initialised as an empty list in the `dvir` namespace with `dvir::set("fonts", vector("list", 255))`. To stop this initialisation happening by default, the following code was put in its place.

```r
if (dvir::get("initFonts") || is.null(dvir::get("fonts")))
  dvir::set("fonts", vector("list", 255))
```

Note in this new statement the logical variable `initFonts`, stored in the `dvir` namespace. This lets a user choose whether to intialise the font cache on a `grid.latex()` call. If `initFonts` is `FALSE` or the font list does not exist, for example after the package is loaded, the font list is initialised. Otherwise, the existing font cache remains. An argument, `initFonts`, was added to `grid.latex()` with its default value being equal to the global option `dvir.initFonts`.

In a DVI file, each font definition gives the font a identification number, for example `fontnum=17`. In all the DVI files examined as part of this project, this font identification number is the same when that particular font is used again in another `grid.latex()` call. This number is used as the index of the fonts list where its information is stored.

When the font sweep passes a font definition in the DVI file it checks first whether the index of the font list matching the font number of the new font contains any existing font information:

- If no, then the font information is written to the font list as usual.
- If yes, then it checks if the existing font information is the same as the font information in the DVI file.
  - If yes, do nothing.
  - If no, overwrite the font with the new font information.

The result of this is that across multiple `grid.latex()` calls requiring the same fonts, any future uses of that font are much faster after the font has been used once. Importantly it also has a "fail safe" where if there happens to be a case where the font number was used with a different font in a previous `grid.latex()` call, the font cache is overwritten with the new font so the correct font will be used.

The updated font list is stored in the `dvir` environment to be retrieved the next `grid.latex()` finds a font definition during a font sweep.

The only weakness of this process is if in a *single* DVI file different fonts are defined with the same font number. In this case the second font will be stored in the font cache in that index and so will be used in place of the first font. No instances of this occurring were discovered during this project.

To examine the effect of these changes on code speed we performed profiling as was detailed in the previous section. For the simple example the time for the font sweep has reduced by 71%, with `grid.latex()` overall taking 45% less time. Similar results can be seen in for the example in figure 9 - a 76% and 46% decrease in the font sweep and `grid.latex()` respectively.

5.1 Profiling environment specifications

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

The profiling results are also dependent on the computing environment with which the `dvir` package is used.

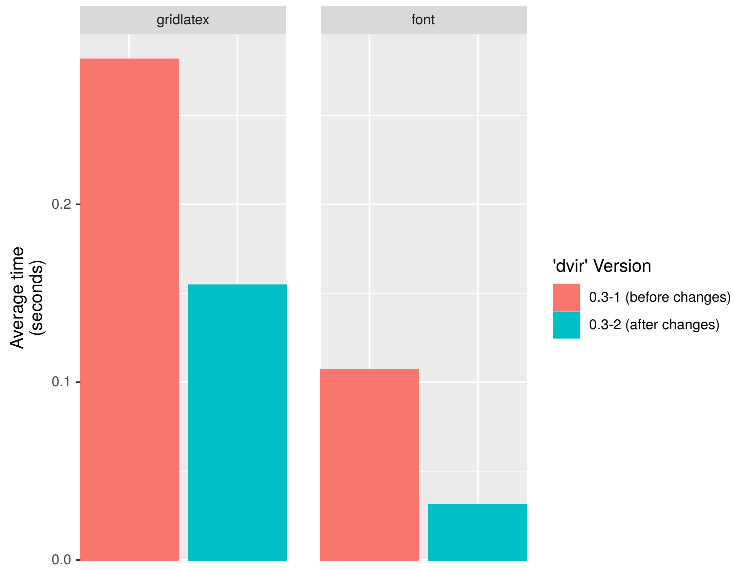

Figure 15: The average time spent in the `grid.latex()` function and font sweep, over 20 runs of the simple example.

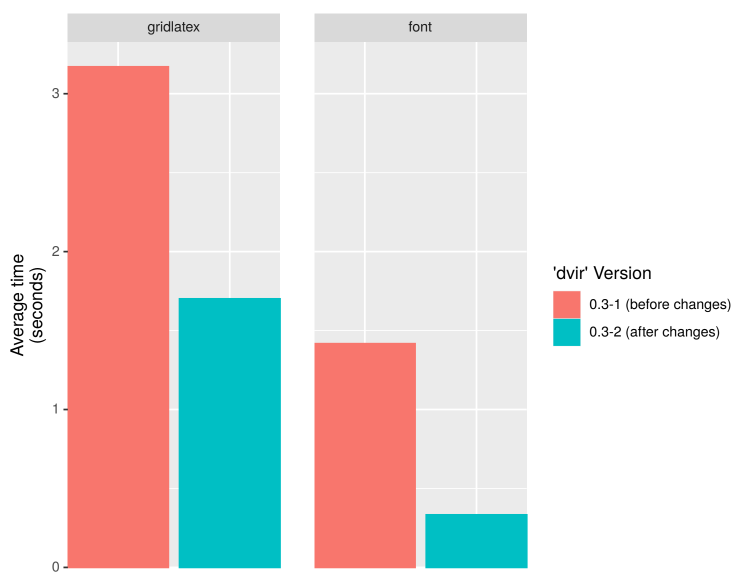

Figure 16: The average time spent in the `grid.latex()` function and font sweep, over 20 runs of the complicated example.

The profiling results in this report, in this and the previous 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

6 Linear gradient fills

6.1 Ti*k*Z and `dvir`

Ti*k*Z (Tantau 2021) is a TEX package that allows drawing of pictures and diagrams in TEX documents.

```
\begin{tikzpicture}
\path (0, 1) node[circle,fill=red!40,draw,thick] (x) {$\mu$}
      (3, 0) node[circle,fill=blue!40,draw,thick] (y) {$\dfrac{\mu^2}{2}$};
\draw[->] (x) .. controls (1, 0.5) and (2, 0.5).. (y);
\end{tikzpicture}
```

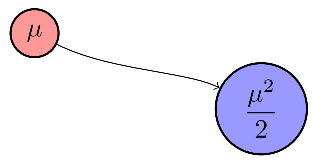

Figure 17: A picture made with Ti*k*Z.

The original DVI specification only needed to account for text, typesetting and the most basic of rectangles 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 Ti*k*Z 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 Ti*k*Z driver. The `dvir` package (Murrell 2020a) includes its own Ti*k*Z driver to translate the drawing instructions into a form useful to draw the things with R grid graphics.

Some Ti*k*Z 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 (Murrell 2020b), on which `dvir` is built.

```
tikzCode <-
  paste("\\path (0, 1) node[circle,fill=red!40,draw,thick] (x) {$\\mu$}",
        "       (3, 0) node[circle,fill=blue!40,draw,thick] (y) {$\\dfrac{\\mu^2}{2}$};",
        "\\draw[->] (x) .. controls (1, 0.5) and (2, 0.5).. (y);",
        sep = "\n")
tikzPreamble <- gsub("\\\\usepackage\\{tikz}",
                     "\\\\usepackage{tikz}\n\\\\usepackage{amsmath}",
                     tikzpicturePreamble())
grid.tikzpicture(tikzCode, preamble = tikzPreamble)
```

The use of `grid.tikzpicture()` is demonstrated in figure 18, recreating the diagram from figure 17. The alteration to the preamble is once again only necessary due to the use of `\dfrac` from the `amsmath` package.

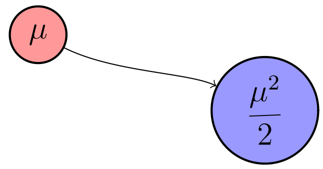

Figure 18: Using `grid.tikzpicture()` to make the earlier diagram.

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

6.2 Implementing Ti*k*Z linear gradient fills in `dvir`

The following steps are required to implement these Ti*k*Z fills in `dvir`:

1. Add the fill information (like the gradient colours and their locations) 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` Ti*k*Z driver file to include information about the gradient and pattern fills. As the `dvir` Ti*k*Z driver file is based on the SVG Ti*k*Z driver file, the SVG support for Ti*k*Z fills was used as a base to edit to make it specific to `dvir`.

The information we require for the gradient fills from Ti*k*Z 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 Ti*k*Z in the DVI file so we can pass it to `dvir`.

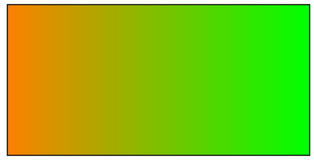

Figure 19: A rectangle with a linear gradient fill using Ti*k*Z.

Let us consider a simple example, a rectangle with an orange to green linear gradient fill, like that in figure 19.

The following is an extract of the DVI file when the rectangle above is generated using the SVG DVI driver included with common TEX distributions, `pgfsys-dvisvgm.def`. It has been edited slightly for readability by adding line breaks and removing the line break character string `{?nl}`.

```
xxx1        k=67
            x=dvisvgm:raw <g transform="matrix(1,0,0,1,56.90549,28.45274)">
xxx1        k=67
            x=dvisvgm:raw <g transform="matrix(2.26802,0,0,1.134,0.0,0.0)">
xxx1        k=66
            x=dvisvgm:raw <g transform="matrix(0.0,1.0,-1.0,0.0,0.0,0.0)">
xxx4        k=425
            x=dvisvgm:raw  <linearGradient id="pgfsh2" gradientTransform="rotate(90)">
                           <stop offset=" 0.0"  stop-color=" rgb(0.0%,100.0%,0.0%) "/>
                           <stop offset=" 0.25" stop-color=" rgb(0.0%,100.0%,0.0%) "/>
                           <stop offset=" 0.5"  stop-color=" rgb(50.0%,75.0%,0.0%) "/>
                           <stop offset=" 0.75" stop-color=" rgb(100.0%,50.0%,0.0%) "/>
                           <stop offset=" 1.0"  stop-color=" rgb(100.0%,50.0%,0.0%) "/>
                           </linearGradient>
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"/>
```

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. 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. There are several TEX macros that were defined to do this. These are based on the `pgfsys-common-svg.def` and `pgfsys-dvisgm.def` SVG drivers that come with most TEX distributions.

A "wrapper" macro of what to do when a gradient fill is requested. Within this, the definition of the fill is created and sent to the DVI file, followed by a rectangle with a fill specified by that definition. For clarity, a line stating when the gradient fill is defined and then again when it is used has been added to the DVI file but these can be removed.

```
\def\pgfsys@shadinginsidepgfpicture#1{%
  #1%
  \pgfsysprotocol@literal{SHADING BEING DEFINED:
                          ShadDefID = \the\pgf@sys@dvir@objectcount}%
  \pgf@sys@dvir@sh@defs%
  \pgf@process{\pgf@sys@dvir@pos}%
  \pgf@xa=-.5\pgf@x%
  \pgf@ya=-.5\pgf@y%
  \pgfsysprotocol@literal{
      <g transform="translate(\pgf@sys@tonumber{\pgf@xa},\pgf@sys@tonumber{\pgf@ya})">}%
  \pgfsysprotocol@literal{SHADING BEING USED: ShadDefID = \the\pgf@sys@dvir@objectcount}
  \pgf@sys@dvir@sh%
}
```

In basic cases, a horizontal linear gradient is defined as a vertical gradient with a 90 degree rotation which is why this is called "vert" shading, as it is in the SCG driver file. The definition and use of the gradient specified above are collated here. The stop positions (proportion along the length of the gradient for which a colour specified) and their respective colours are collated and the literal text to build the gradient definition and rectangle using that definition. The biggest difference from the SVG driver was that R needs a vector of stops and a separate vector of colours as arguments to `linearGradient()`. SVG specifies the position and colour of each stop together.

```
\def\pgfsys@vertshading#1#2#3{%
  {%
    \pgf@parsefunc{#3}%
    \global\advance\pgf@sys@dvir@objectcount by1\relax%
    \pgf@sys@dvir@shading@stops%
    \pgf@sys@dvir@shading@stopcolours%
    \expandafter\xdef\csname @pgfshading#1!\endcsname{%
      \def\noexpand\pgf@sys@dvir@sh@defs{
          \noexpand\pgfsysprotocol@literal{\pgf@sys@dvir@thestops}}%
      \def\noexpand\pgf@sys@dvir@sh{\noexpand\pgfsysprotocol@literal{<rect
        width="\pgf@sys@tonumber{\pgf@y}"
        height="\pgf@sys@tonumber{\pgf@x}"
        style="fill:url(\noexpand\#pgfsh\the\pgf@sys@dvir@objectcount);
          stroke:none"/>\noexpand\pgfsys@dvir@newline}}%
      \def\noexpand\pgf@sys@dvir@pos{\noexpand\pgfpoint{\the\pgf@y}{\the\pgf@x}}%
    }%
  }%
}
```

This macro allows us to iteratively add stop positions or colours to the ones we have already gathered.

```
\let\pgf@sys@dvir@thestops=\pgfutil@empty
\def\pgf@sys@dvir@addtostops#1{%
  \edef\pgf@temp{#1}%
  \expandafter\expandafter\expandafter\def
  \expandafter\expandafter\expandafter\pgf@sys@dvir@thestops
  \expandafter\expandafter\expandafter{
      \expandafter\pgf@sys@dvir@thestops\expandafter\space\pgf@temp}%
}
```

The following macros process all the stop locations, collating them into an R friendly vector suitable to parse to `linearGradient()`.

```
\def\pgf@sys@dvir@shading@stops{%
  % Step 1: Compute 1/\pgf@sys@shading@end@pos
  \pgf@x=\pgf@sys@shading@end@pos\relax%
  \c@pgf@counta=\pgf@x\relax%
  \divide\c@pgf@counta by4096\relax%
  % Step 2: Insert stops locations
  \pgf@sys@dvir@addtostops{stops=(}%
  \expandafter\pgf@sys@dvir@shading@dostoplocations\pgf@sys@shading@ranges%
  % dummy for end:
  {{\pgf@sys@shading@end@pos}{\pgf@sys@shading@end@pos}}%
  \pgf@sys@dvir@addtostops{)}%
}
```

```
\def\pgf@sys@dvir@shading@dostoplocations#1{%
  \edef\pgf@test{#1}%
  \ifx\pgf@test\pgfutil@empty%
  \else%
    \expandafter\pgf@sys@dvir@shading@dostoplocation\pgf@test%
    \expandafter\pgf@sys@dvir@shading@dostoplocations
  \fi%
}

\def\pgf@sys@dvir@shading@dostoplocation#1#2#3#4{%
  % #1 start pos
  % #2 end pos
  % #3 start rgb
  % #4 end rgb
  \pgf@x=#1%
  \pgf@x=16\pgf@x%
  \divide\pgf@x by \c@pgf@counta\relax%
  \expandafter\pgf@sys@dvir@addtostops{\pgf@sys@tonumber\pgf@x}%
}
```

Similar to the above, these collate all the colours of the stops into an R friendly vector.

```
\def\pgf@sys@dvir@shading@stopcolours{%
  % Step 1: Compute 1/\pgf@sys@shading@end@pos
  \pgf@x=\pgf@sys@shading@end@pos\relax%
  \c@pgf@counta=\pgf@x\relax%
  \divide\c@pgf@counta by4096\relax%
  % Step 2: Insert stops RGB colours
  \pgf@sys@dvir@addtostops{, colours=(}%
  \expandafter\pgf@sys@dvir@shading@dostopcolours\pgf@sys@shading@ranges%
  % dummy for end:
  {{\pgf@sys@shading@end@rgb}{\pgf@sys@shading@end@rgb}{\pgf@sys@shading@end@rgb}}%
  \pgf@sys@dvir@addtostops{)}%
}

\def\pgf@sys@dvir@shading@dostopcolours#1{%
  \edef\pgf@test{#1}%
  \ifx\pgf@test\pgfutil@empty%
  \else%
    \expandafter\pgf@sys@dvir@shading@dostopcolour\pgf@test%
    \expandafter\pgf@sys@dvir@shading@dostopcolours%
  \fi%
}

\def\pgf@sys@dvir@shading@dostopcolour#1#2#3#4{%
  % #1 start pos
  % #2 end pos
  % #3 start rgb
  % #4 end rgb
  \expandafter\pgf@sys@dvir@shading@dorgb#3%
}
```

This is a helper function to return formatted RGB values.

```
\def\pgf@sys@dvir@shading@dorgb#1#2#3{%
  \pgf@sys@dvir@color@rgb#1,#2,#3\relax%
```

```
  \pgf@sys@dvir@addtostops{\pgf@sys@dvir@prepared}%
}
```

A counter is defined in order to give each gradient definition unique identifier.

```
\newcount\pgf@sys@dvir@objectcount
```

The result of these changes mean that the linear gradient definition is specified in the DVI file in a form usable by R, sans some commas.

```
xxx1         k=118
             x=dvir::  stops=( 0.0 0.25 0.5 0.75 1.0 ) ,
                       colours=( rgb(0,1,0) rgb(0,1,0) rgb(0.5,0.75,0) rgb(1,0.5,0)
                                 rgb(1,0.5,0) )
```

Compare this with the linear gradient definition from earlier:

```
xxx4         k=425
             x=dvisvgm:raw  <linearGradient id="pgfsh2" gradientTransform="rotate(90)">
                            <stop offset=" 0.0" stop-color=" rgb(0.0%,100.0%,0.0%) "/>
                            <stop offset=" 0.25" stop-color=" rgb(0.0%,100.0%,0.0%) "/>
                            <stop offset=" 0.5" stop-color=" rgb(50.0%,75.0%,0.0%) "/>
                            <stop offset=" 0.75" stop-color=" rgb(100.0%,50.0%,0.0%) "/>
                            <stop offset=" 1.0" stop-color=" rgb(100.0%,50.0%,0.0%) "/>
                            </linearGradient>
```

Unfortunately this was the extent this project could explore implementing Ti*k*Z gradient fills in `dvir`. You may notice in both the SVG and `dvir` DVI output that the colour at stops 0 and 0.25 are the same, as is the colour at stops 0.75 and 1. This is related to a larger issue of Ti*k*Z using a number of transformations on the "simple" linear gradient definition above which is then clipped to the shape the fill is for. These transformations would require some very detailed work to translate them into a set of instructions to perform the same in R before they are used in `linearGradient()`.

An additional complication is that Ti*k*Z will create shapes with these fills by first drawing the shape with a transparent fill and then drawing another shape with the gradient fill clipped to the size of the first shape. R however requires the the fill information as an argument when drawing the shape. In R, any transformations of the gradient, as Ti*k*Z does, will need to be done *before* it is parsed to the drawing function.

Radial gradient fills have similar problems with how they are manipulated by Ti*k*Z, however the work done so far on linear gradients could be easily replicated for them.

This work is still within step 1 of the workflow in section 6.2. To continue with the implementation, aside from the above obstacles, `dvir` would need to do another sweep of the DVI file, like the font sweep, to achieve step 2. After any transformations or manipulations are made, the gradient definition can be called by a unique identifier to parse the stored fill definitions to the drawing function used as in step 3.

7 Text baselines

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 y or j, have a "descender". A descender is a part of a character that sits below the baseline.

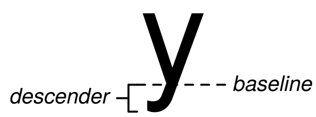

Figure 20: The baseline and descender of the letter y.

The `grid.text()` function 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. Unfortunately, `dvir` does not account for text baselines and will do any alignment in relation to bounding box of the text, which, in the case of figure 21, is at the bottom of the descenders.

```
texText <- "{\\LARGE changing to \\LaTeX{}}"
grid.text("An example of ",
          y = 0.1,
          just = c("right", "bottom"),
          gp = gpar(fontsize = 18))
grid.latex(texText, y = 0.1, just = c("left", "bottom"))
grid.lines(x = c(0.1, 1), y = 0.1, gp = gpar(lwd = 0.5))
```

Figure 21: Combining `grid.text()` and `grid.latex()`.

To fix this, `dvir` needs to obtain or calculate a value for the baseline for any given piece of text to use as an offset for the bounding box alignment when it is rendering the text. From this point on, this offset will be referred to as a baseline value. All of `dvir`'s information comes from the DVI file so it either needs to extract a baseline value from the DVI file itself, or calculate it *from* information in the DVI file. Unfortunately plain DVI files do not state a baseline value, rather they only detail specific placement of each character. As such, some possible methods to obtain a baseline value were explored. These methods are referred to as algorithms from here on due to their heuristic nature.

7.2 Implementation

To explore the algorithms detailed below and evaluate their usefulness an R function, `baselines()`, has been created. This function can be found in the appendix in the file `algorithms.R`. This function takes several arguments, including the baseline selection algorithm as detailed below, any additional information needed for that particular algorithm, and the TEX code as you would use with `grid.latex()`. The output of this function is the baseline value or values. These values are returned as `grid` units as they are distances from the bottom of the bounding box of the text to the text's baseline.

Once the baseline values have been calculated the bounding box of the text can be bottom-aligned with the y-value specified but then moved down by the baseline value. This means the baseline of the text, should the algorithms work correctly, will be aligned at the specified y-value.

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

7.3 Potential solutions

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

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.

```
xxx1         k=50
             x=ps::%%HiResBoundingBox: 0 0 14.21944pt 25.85977pt.
```

After this statement there appears to consistently be a downward move equal to the height of the bounding box of text (from the top left of the bounding box to the bottom left), 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 and the baseline value returned is 0.

```r
baselineValue <- baselines(tex = "$x - \\mu$", algorithm = "alex")
baselineValue
```

```
## [1] 127431scaledpts
```

For plain text this algorithm works quite well. Unfortunately for many equations, particularly ones with superscripts or subscripts, the first upward move is *not* to where the baseline is. Note in figure 22 the `alex` algorithm has calculated the baseline of the fifth example to be the bottom of the μ. This demonstrates that this algorithm is only practical when the first character's baseline is the one to be aligned, and in the case of the second example, does not serve much purpose at all.

7.3.2 `dviMoves` algorithm

This is an extension of the `alex` algorithm. Rather than only taking the location after the second vertical cursor 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 should reflect the cursor moving to the

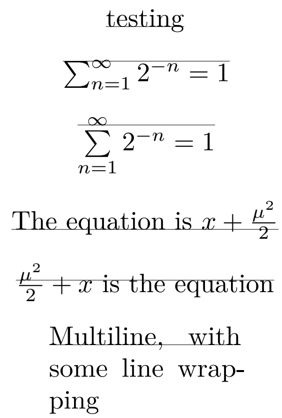

Figure 22: The baselines as calculated by the `alex` algorithm.

baseline of the next character to be typeset. Once again we assume that the first downward move after the "HiResBoundingBox" statement is from the top left to the bottom left of the bounding box so this algorithm only returns the upward and downward moves from there. 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 tracked 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 baseline values returned. Any more than one baseline value being returned means a method for choosing which baseline value is required.
- There are often several up and downward moves in the DVI file between typeset characters so in between the "useful" baseline values there can be some that are duplicates or are not very useful.

To account for these considerations, along with the `dviMoves` algorithm, the `baselines()` function also allows a choice of method to select a *single* baseline out of the many returned by the algorithm.

7.3.2.1 `dviMoves` selection method `all`

This selection method will return all the baseline values as determined by the algorithm. This is useful for drawing the text with multiple lines representing all the baseline values calculated from this method.

7.3.2.2 `dviMoves` selection method `index`

When this selection method is chosen, another argument to the function, `dviMovesSelection`, is used to specify a numeric index. The baseline value corresponding to that index is returned.

7.3.2.3 `dviMoves` selection method `bottomUp`

This is similar to the `index` method but the baseline values are first ordered from smallest to largest, before using the `dviMovesSelection` argument to select the index of the baseline value to return.

7.3.2.4 `dviMoves` selection method `nextChars`

This method will return the first baseline value just after a specified character in the argument `dviMovesSelection`. Characters are specified by a number, entered as a character string. This method is limited for several reasons:

- It cannot return a baseline value for any instance of a character after the first instance of that character
- If multiple cursor moves are made before a character is drawn, the earlier moves are still recorded but with an empty character label
- Some characters are hard to type on a "normal" keyboard, for example μ
- The DVI file does not actually contain the character, but rather the numeric index of it in the particular font. This is why a number has to be used to select which character it is.

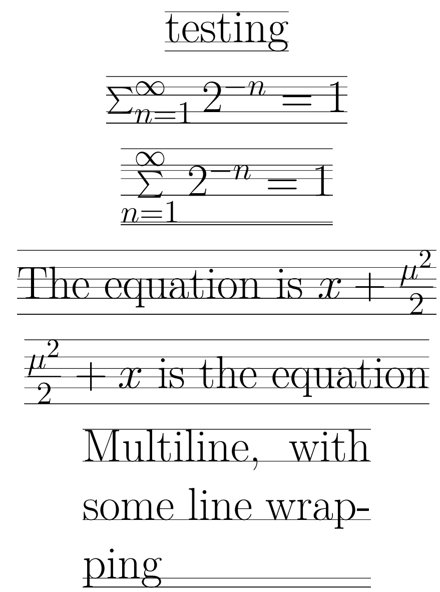

Figure 23: The baselines as calculated by the `dviMoves` algorithm.

This algorithm performs very well. In figure 23 nearly every part of all examples has a baseline value associated with it, including the superscripts of math equations and every line of the multiline text. This means alignment would be possible with these parts of the examples.

7.3.3 `preview` algorithm

This algorithm uses the `preview` package (Kastrup 2017) in LaTeX. Gieseking (2017) describes the use of the algorithm for `dvisvgm`, and it has been implemented here as it has in `matplotlib.texmanager` (Hunter 2007).

When this algorithm is selected, the following code is added to the header of the DVI file.

```
\usepackage[active,showbox,tightpage]{preview}
\def\\showbox#1%
{\immediate\\write16{MatplotlibBox:(\\the\\ht#1+\\the\\dp#1)x\\the\\wd#1}}
```

This results in the following line being printed in the log file.

`MatplotlibBox:(8.04175pt+3.00005pt)x62.81718pt`

The baseline value in this case is the value just after the `+` and can be obtained by searching the log file for this line starting with `MatplotlibBox:` and returning the value just after the `+`.

testing

$\sum_{n=1}^{\infty} 2^{-n} = 1$

$$\sum_{n=1}^{\infty} 2^{-n} = 1$$

The equation is $x + \frac{\mu^2}{2}$

$\frac{\mu^2}{2} + x$ is the equation

Multiline, with
some line wrap-
ping

Figure 24: The baselines as calculated by the `preview` algorithm.

This algorithm also performs very well. In all examples in figure 24 the natural baseline of the text has been identified and so this algorithm would be suitable in most use cases. For the multiline text, the second line of text has been identified as having "the" baseline.

7.3.4 `dvipng` algorithm

The `dvipng` program (Larsson 2020) turns DVI files into PNG images. An option of `dvipng` is `--depth` which returns the baseline value in pixels.

The system call `dvipng --depth test.dvi` returns `depth=4` within its output so after recording the output, a simple search for `depth=` will find the baseline value.

testing

$\sum_{n=1}^{\infty} 2^{-n} = 1$

$$\sum_{n=1}^{\infty} 2^{-n} = 1$$

The equation is $x + \frac{\mu^2}{2}$

$\frac{\mu^2}{2} + x$ is the equation

Multiline, with
some line wrap-
ping

Figure 25: The baselines as calculated by the `dvipng` algorithm.

It may not be obvious in printed form in figure 25 but the only example with which the `dvipng` algorithm has selected a "good" baseline value is one of the equations. The other examples have baseline values which are at least slightly off. This is likely a side effect of this algorithm returning the baseline value in (whole) pixels which does not lend itself well to this detailed application.

7.4 Discussion of these algorithms

Of the algorithms considered, the `dviMoves` algorithm performed best. While these algorithms (except `dvipng`) perform well for some or most of the examples, the `dviMoves` algorithm performs well for *all* the examples. Notably, it gives the option to align the baseline of *any* line of multiline text or with almost any character in a mathematical equation, whether it be of a different size, part of a fraction, or a superscript or subscript.

The difficulty with the algorithm is how to choose a *single* baseline value from all the ones calculated. It is possible to manually check them all the baseline values to find the right one but an automated selection method would be desirable. As the `preview` algorithm calculated baseline values for all examples except the multiline text, the algorithms would work well together - an implementation of this functionality in `dvir` could calculate the baseline values with both the `dviMoves` and `preview` algorithms with the default selection being the baseline value from `dviMoves` that is closest to the baseline value calculated by the `preview` algorithm. This could be overridden to select another baseline value calculated by `dviMoves`.

8 Discussion

TeX and its extensions have more functionality than could realistically be added to R but by targeting the most useful features of TeX, as `dvir` did originally and as has been continued in this project, many use cases are accommodated.

The speed improvement of the package has made the experience using `dvir` better when creating graphics with many `grid.latex()` calls and also when *developing* graphics with `grid.latex()` which require rerunning code with small tweaks often to get the perfect graphical result. The progress on aligning `grid.latex()` text output with its baseline means it is possible to professionally display text and equations when they are integrated with other text or graphical features.

Of course, there are still many things which could improve `dvir` in future. Now that the font sweeping by `dvir` is much more efficient other parts of the code are becoming the "slowest". In particular the reading and writing to disk, of the DVI file for example, takes more time now relative to the rest of the code.

Implementation of a text baseline algorithm into `grid.latex()` itself to align text would also be a useful step forward. In the instance of the `dviMoves` algorithm, care is needed when choosing a method to determine the baseline value and should allow a user to manually override the choice as well. Additional testing of the algorithms could also be done to consider more advanced use cases, for example multiline text with mathematical equations and altered line spacing. Work on the Ti*k*Z gradient fills could also progress as R expands its support of the gradient fills.

9 References

Chang, Winston. 2014. *Extrafont: Tools for Using Fonts*. https://cran.r-project.org/package=extrafont.

Chang, Winston, Alexej Kryukov, and Paul Murrell. 2014. *Fontcm: Computer Modern Font for Use with Extrafont Package*. https://cran.r-project.org/package=fontcm.

Gieseking, Martin. 2017. "dvisvgm: Generating Scalable Vector Graphics from DVI and EPS Files." *TUGboat* 38 (3). https://tug.org/tugboat/tb38-3/tb120gieseking.pdf.

Hunter, John. 2007. "Matplotlib: A 2d Graphics Environment." *Computing in Science & Engineering* 9 (3). IEEE Computer Soc: 90–95. doi:10.1109/MCSE.2007.55.

Kastrup, David. 2017. *The preview Package for LaTeX: Version 12.3*. https://ctan.org/pkg/preview.

Knuth, Donald E. 1986. *The Texbook*. Addison-Wesley Professional.

Larsson, Jan-Åke. 2020. *dvipng - a DVI-to-PNG Translator: Version 1.17*. https://ctan.org/pkg/dvipng.

Murrell, Paul. 2018. "Revisiting Mathematical Equations in R: The 'dvir' Package." 2018-08. Department of Statistics, The University of Auckland.

———. 2020a. "Adding TikZ Support to 'dvir'." 2020-05. Department of Statistics, The University of Auckland. doi:http://dx.doi.org/10.17608/k6.auckland.13283900.

———. 2020b. "Catching up with R Graphics." 2020-04. Department of Statistics, The University of Auckland. doi:http://dx.doi.org/10.17608/k6.auckland.12649751.

Murrell, Paul, and Ross Ihaka. 2000. "An Approach to Providing Mathematical Annotation in Plots." *Journal of Computational and Graphical Statistics* 9 (3). Taylor & Francis: 582–99. doi:10.1080/10618600.2000.10474900.

R Core Team. 2021. *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. https://www.r-project.org/.

Tantau, Till. 2021. *The Tikz and Pgf Packages: Manual for Version 3.1.9a*. https://ctan.org/pkg/pgf .

10 Appendix

Files related to this project can be found on GitHub, via the link in the Executive Summary.