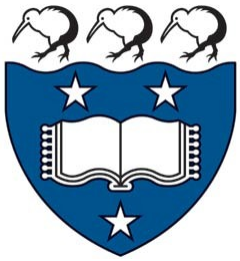
Improvements to the 'dvir' Package

Alexander van der Voorn



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

Contents

1 Executive summary

. . . 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)
```

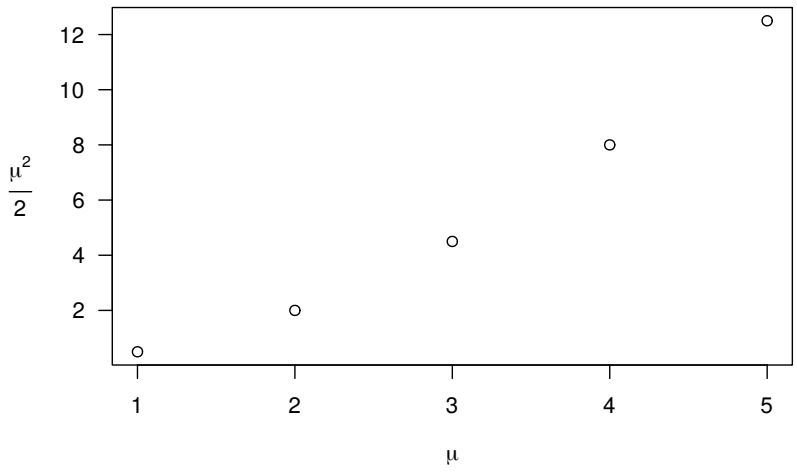


Figure 1: A plot with axis labels made using 'expression()'.

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 2.

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

Figure 2: The y-axis label from above, when 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. 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("$x - \\mu$")
```

$$x - \mu$$

Figure 3: 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.

   - [**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")
```

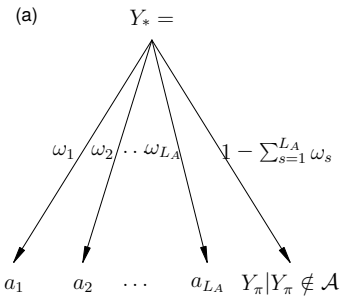


Figure 4: A more complicated example using ‘dvir‘

The example in figure **??** 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.

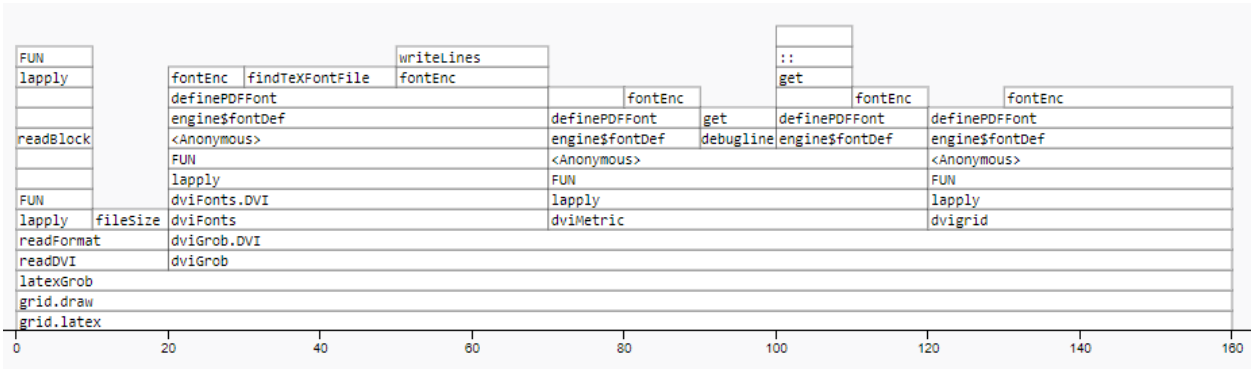


Figure 5: Screenshot of `profvis::profvis()` output for the code `grid.latex("$x - \\mu$")` in `dvir` version 0.2-1.

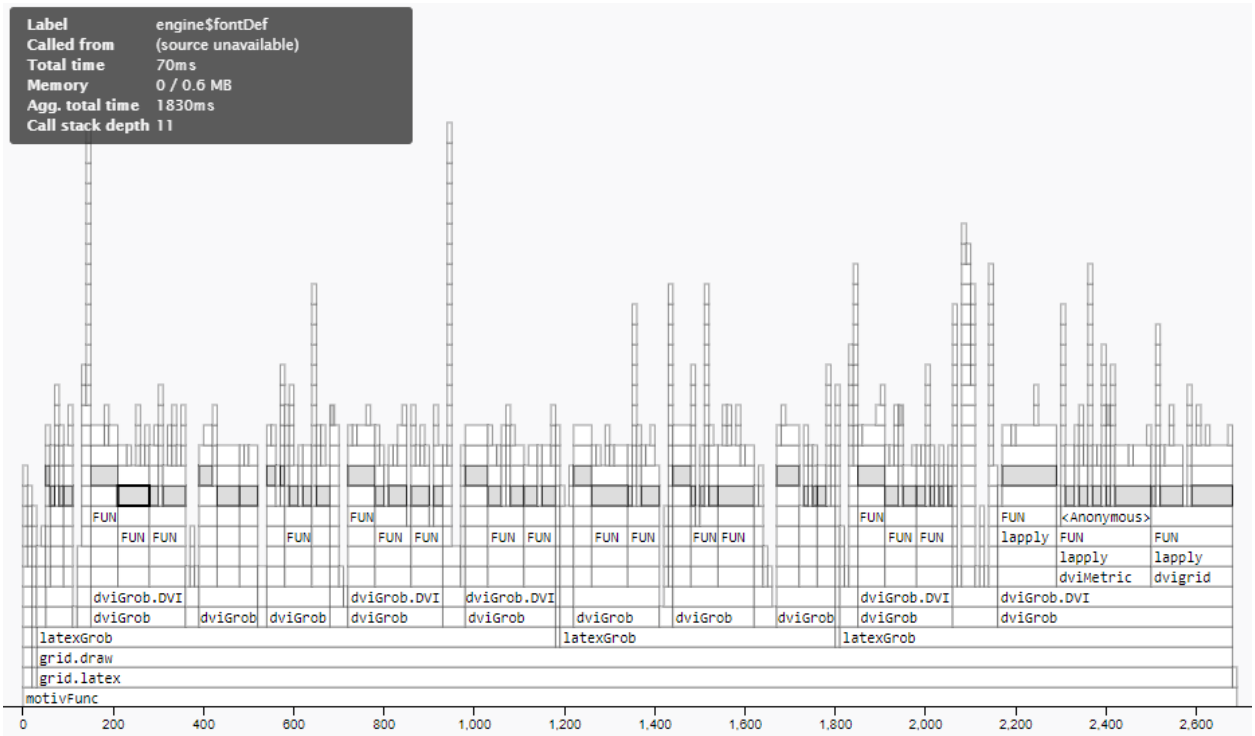


Figure 6: Screenshot of `profvis::profvis()` output for the code creating figure 4 highlighting the time spent in `engine$fontDef`.

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 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.

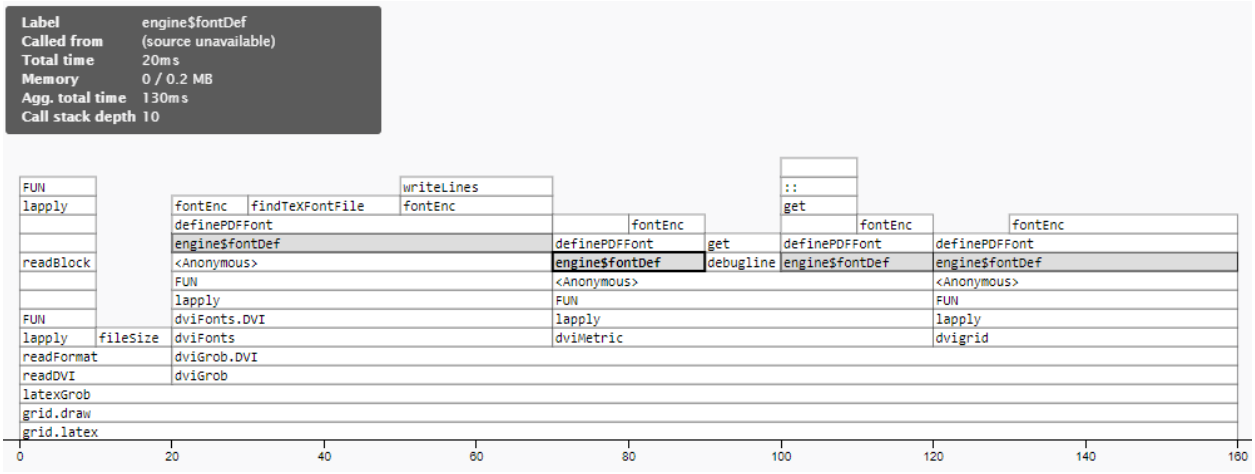


Figure 7: Screenshot of `profvis::profvis()` output for the code `grid.latex("$x - \\mu$")` in `dvir` version 0.2-1, highlighting `engine$fontDef`.

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.

The following code is taken from the the `dvir` package, showing 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_font_def
grid_op_243 <- op_font_def
```

The following code shows 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.

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

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. 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 we recorded the time to run our examples 20 times, after an initial run to compile the package after it was loaded.

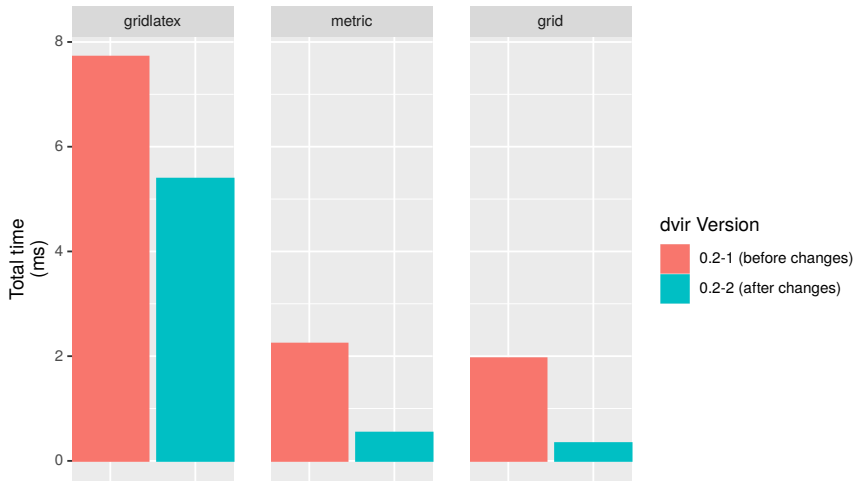


Figure 8: The total time spent in the ‘grid.latex()‘ function, metric sweep and grid sweep before and after these changes, over 20 runs of our simple example.

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 @ref{fig:speedUp1Yee} the metric and grid sweeps have decreased by 83% and 89% respectively, with `grid.latex()` overall taking 47% less time.

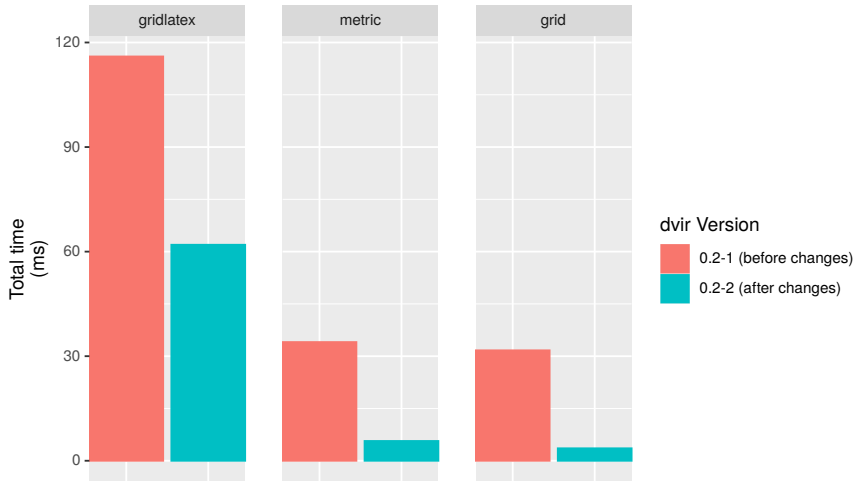


Figure 9: The total time spent in the 'grid.latex()' function, metric sweep and grid sweep before and after these changes, over 20 runs of our complicated example.

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)]

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 call stack) will give different results every time it is done.

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

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

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

The original DVI specification only needed to account for text and 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 includes its own Ti*k*Z driver to translate the drawing instructions into a form useful to draw the things with R grid graphics [reference Paul dvir Ti*k*Z report].

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, on which `dvir` is built.

- [**Example:** replicate one of the above examples in R]
- [**Example:** Ti*k*Z radial gradient fill example]
- [**Example:** Make same Ti*k*Z 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 Ti*k*Z 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 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

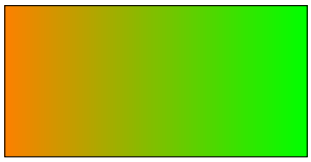
3. Add the fill information when drawing the shape in R

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`.

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



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 by adding line breaks.

```
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. 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)">{?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}
```

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

- [Example - showing a letter, with the baseline marked and also showing what part is the "descender" part]

`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. Unfortunately, `dvir` does not account for text baselines and will do any alignment with the bounding box of the text.

- [Example - combining `grid.text()` and `grid.latex()` on same y-value to show the difference between them]
- [Example - more complicated example, say with several pieces of multi-line text (only if I can demonstrate later that my function works for aligning them!)]

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 extract 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 explore some possible methods to obtain a baseline value. 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 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 distance, or in some case distances, from the bottom of the bounding box of the text to the possible baseline value. These distances are returned as `grid` units.

Once the baselines have been calculated the bounding box of the text can be bottom aligned with the y-value specified but then moved down by the value of the baseline. This means the baseline of the text will be 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 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 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 a value of 0 is returned as the baseline.

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

```
source("../Appendices/algorithms.R")
baselineValue <- baselines(tex = "$x - \\mu$",
                           algorithm = "alex")
baselineValue
```

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

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

test

testing

var

varying

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

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

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

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

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

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

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

Paragraph, with

some line wrapping!

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 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.

- [Example: Show examples from start of this section with baselines being used to make them aligned!]

7.5 Next steps to integrate with `dvir` package

8 Conclusion/summary/next steps

. . . goes here. . .