\[ p(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} \]

\[
\int_{-2}^{2} p(x) \, dx \approx 0.954
\]
The **tikzDevice** Package

http://r-forge.r-project.org/projects/tikzdevice

Charlie Sharpsteen and Cameron Bracken

Version GitHub Dev-**Beta**  November, 2011

Contents

1 Introduction 1

2 Acknowledgements 1

I Usage and Examples 2

3 Loading the Package 2

3.1 R Options That Affect Package Behavior 3

4 The **tikz()** Function 6

4.1 Description 6

4.2 Usage 6

4.3 Font Size Calculations 7

4.4 UTF-8 Output 8

4.5 Examples 8

4.5.1 Default Mode 8

4.5.2 **bareBones** Mode 10

4.5.3 **standAlone** Mode 12

4.5.4 **console output** Mode 13

4.5.5 Getting around special **\LaTeX** characters 13

4.5.6 Using the Global Options: An **Xe\LaTeX** example 13

4.5.7 Annotating Graphics with TikZ Commands 16

4.5.8 **tikz()** vs. **pdf()** for **plotmath** symbols and Unicode characters 18

5 The **getLatexCharMetrics()** and **getLatexStrWidth()** Functions 19

5.1 Description 19

5.2 Usage 19

5.3 Examples 19

II Installation Guide 21

6 Obtaining a **\LaTeX** Distribution 21

6.1 Windows 21

6.2 UNIX/Linux 21

6.3 Mac OS X 21
7 Installing Ti\textit{k}Z and Other Packages 22
  7.1 Using a \texttt{\LaTeX} Package Manager ........................................ 22
  7.2 Manual Installation ................................................................. 22

III Package Internals 24

8 Introduction and Background 24

9 Anatomy of an \texttt{R} Graphics Device 25

10 Calculating Font Metrics 25
  10.1 Character Metrics ................................................................. 26
  10.2 Calling \texttt{R} Functions from \texttt{C} Functions ....................... 27
  10.3 Implementing a System Call to \texttt{\LaTeX} ................................. 29

11 On the Importance of Font and Style Consistency in Reports 33

12 The \texttt{pgfSweave} Package and Automatic Report Generation 33
1 Introduction

The `tikzDevice` package allows for R graphics output in a native \LaTeX{} format. That is, the `tikz()` function produces plain-text files that can be interpreted using \LaTeX{}, a package for \LaTeX{}. These files can be directly included in \LaTeX{} documents by way of the `\input{}` statement. Allowing \LaTeX{} to handle both typesetting and figure composition bestows the resulting document with a clean, unified look as there are no discontinuities in the size and selection of fonts used in the text and those used in the figures.

This document is divided into three parts. The first part describes the functions that the package makes available to the R user and provides examples of their capabilities. Besides the R environment, use of the \LaTeX{} device requires the user to have a working \LaTeX{} compiler along with an installed version of the \LaTeX{} package– version 2.00 or greater. The second part of this documentation offers suggestions on how to get these tools working properly.

The third part of the documentation is intended for those who are curious as to the details of how this package has been implemented. It attempts to explain how the \LaTeX{} package does the things that it does and why it chooses to do them that way. The authors have attempted to write this part of the documentation in a way that is accessible to users as well as developers. This has been done in the hope that this project may serve as a case study in creating an R graphics device. This part of the documentation may also help those considering undertaking the transition from casual package-building to full-on hacking of the R internals.

2 Acknowledgements

This package would not have been possible without the hard work and ingenuity of many individuals. This package straddles the divide between two great open source communities– the R programming language and the \LaTeX{} typesetting system. It is our hope that this work will make it easier to leverage the strengths of both systems.

First off, we would like to thank the R Core Team for creating such a wonderful, open and flexible programming environment. Compared to other languages we have used, creating packages and extensions for R has always been a liberating experience.
This package started as a fork of the PicTeX device, created by Valerio Aimale. Without access to such a concise, compact example of implementing a R graphics device we likely would have abandoned the project in its infancy. We would also like to thank Paul Murrel for all of his work on the R graphics system and especially for his research and documentation concerning the differences between the font systems used by TeX and R.

This package also owes its existence to Friedrich Leisch’s work on the Sweave system and Roger D. Peng’s cacheSweave extension. These two tools got us interested in the concept of Literate Programming and development of this package was driven by our desire to achieve a more seamless union between our reports and our code.

The performance of this package is also enhanced by the database capabilities provided by Roger D. Peng’s filehash package. Without this package, the approach to calculating font metrics taken by the tikzDevice would be infeasible.

Last, but certainly not least, we would like to thank Till Tantau, Mark Wibrow and the rest of the PGF/TikZ team for creating the LATEX graphics package that makes the output of this device meaningful. We would also like to express deep appreciation for the excellent documentation that has been created for the TikZ system.

As always, there are many more who have contributed in ways too numerous to list.

Thank you!
—The tikzDevice Team

Part I

Usage and Examples

3 Loading the Package

The functions in the tikzDevice package are made accessible in the R environment by using either the library() or require() functions like so:

```r
require(tikzDevice)
```

Upon loading, the package will search for a usable LATEX compiler. Access to LATEX is essential for the device to produce correct output as the compiler is queried for font metrics several times during device output. For more information on why communication between the device and LATEX is necessary, see Part III. If the search for a compiler is successful the package startup message should look similar to the following:

```
filehash: Simple key-value database (2.0-1 2008-12-19)
tikzDevice: A Device for R Graphics Output in PGF/TikZ Format (v0.3.5)
Checking for a LaTeX compiler...
pdfTeX 3.1415926-1.40.9-2.2 (Web2C 7.5.7)
kpathsea version 3.5.7
...
A working LaTeX compiler was found in:
The R environment variable R_LATEXCMD
```
If a working \LaTeX{} compiler cannot be found, the \pkg{tikzDevice} package will fail to load and a warning message will be displayed:

```{r}
An appropriate \LaTeX{} compiler could not be found.
Access to \LaTeX{} is currently required in order for the
\texttt{TikZ} device to produce output.
```

The following places were tested for a valid \LaTeX{} compiler:

- A pre-existing value of the global option \code{tikzLatex}
- The R environment variable \code{R_LATEXCMD}
- The R environment variable \code{R_PDFLATEXCMD}
- The global option \code{latexcmd}
- The PATH using the command \texttt{latex}
- The PATH using the command \texttt{pdflatex}

...  

Error : .onLoad failed in loadNamespace for \pkg{tikzDevice}
Error: package/namespace load failed for \pkg{tikzDevice}

In this case, \pkg{tikzDevice} has done it’s very best to locate a working compiler and came up empty. If you have a working \LaTeX{} compiler, the next section describes how to inform the \pkg{tikzDevice} package of its location. For suggestions on how to obtain a \LaTeX{} compiler, see Part II.

### 3.1 R Options That Affect Package Behavior

The \pkg{tikzDevice} package is influenced by a number of options that may be set locally in your R scripts or in the R console or globally in a \code{.Rprofile} file. All of the options can be set by using \code{options(<option> = <value>)}. These options allow for the use of custom \code{documentclass} declarations, \LaTeX{} packages, and typesetting engines (e.g. \TeX{}). The defaults, if any for a given option, are shown below the description. The global options are:

- \code{tikzDefaultEngine} Specifies which typesetting engine functions in the \pkg{tikzDevice} package will prefer. Current possible values are \code{pdftex} or \code{xetex} which will respectively trigger the use of the \pdfTeX{} and \xeTeX{} compilers.

```{r}
Choosing the \TeX{} engine

```{r}
\begin{verbatim}
options(tikzDefaultEngine = 'pdftex')
# Or...
options(tikzDefaultEngine = 'xetex')
\end{verbatim}
```

- \code{tikzLatex} and \code{tikzXelatex} Specifies the location of the \LaTeX{} and XeLaTeX compilers to be used by \pkg{tikzDevice}. Setting a default for this this option may help the package locate a missing compiler:
Setting default compilers in `.Rprofile`

```r
options(tikzLatex = '/path/to/latex/compiler')
options(tikzXelatex = '/path/to/xelatex/compiler')
```

tikzMetricsDictionary When using the graphics device provided by `tikzDevice`, you may notice that R appears to “lag” or “hang” when commands such as `plot()` are executed. This is because the device must query the \LaTeX compiler for string widths and font metrics. For a normal plot, this may happen dozens or hundreds of times—hence R becomes unresponsive for a while. The good news is that the `tikz()` code is designed to cache the results of these computations so they need only be performed once for each string or character. By default, these values are stored in a temporary cache file which is deleted when R is shut down. A location for a permanent cache file may be specified:

```r
options(tikzMetricsDictionary = '/path/to/dictionary/location')
```

tikzDocumentDeclaration A string. The documentclass declaration when `standAlone == TRUE` as well as when font metrics are calculated

```r
options(tikzDocumentDeclaration = '\documentclass {article}')
```

Default

tikzFooter A character vector. The footer to be used only when `standAlone==TRUE`

```r
options(tikzFooter = c('\end{document}' ))
```

Default

tikzLatexPackages and tikzXelatexPackages Character vectors. These are the packages which are included when using the `standAlone` option as well as when font metrics are calculated. If you use additional packages that affect fonts, such as mathpazo which provides a Palatino-style font, these packages should be added to this list.

```r
options(tikzLatexPackages = c(  '\usepackage{tikz}',  '\usepackage[active,tightpage]{preview}',  '\PreviewEnvironment{pgfpicture}',  '\setlength\PreviewBorder{0pt}')
```

Default
Adding a package that affects fonts

\begin{verbatim}
options( tikzLatexPackages = c(
  getOption( "tikzLatexPackages" ),
  "\usepackage{mathpazo}"
))
\end{verbatim}

tikzMetricPackages and tikzUnicodeMetricPackages Character vectors. These are the extra packages which are additionally loaded when doing font metric calculations. As you see below, the font encoding is set to Type 1. This is very important so that character codes of \LaTeX{} and R match up. The Unicode metric packages are used when a Xe\TeX{} engine is in use.

\begin{verbatim}
options( tikzMetricPackages = c(
  \"\usepackage[utf8]{inputenc}\",
  \"\usepackage[T1]{fontenc}\",
  \"\usetikzlibrary{calc}\")
)
\end{verbatim}

tikzSanitizeCharacters A character vector of special latex characters to replace. These values should correspond to the replacement values from the tikzReplacementCharacters option.

\begin{verbatim}
options( tikzSanitizeCharacters = c('%','$','}','{','^') )
\end{verbatim}

tikzReplacementCharacters A character vector of replacements for special latex characters. These values should correspond to the values from the tikzSanitizeCharacters option.

\begin{verbatim}
options( tikzReplacementCharacters = c(\% ','\$',' ',' ','^') )
\end{verbatim}

tikzRasterResolution When tikz is requested to add a raster to a graphic, the raster is written to a PNG file which is then included into the LaTeX code. This option controls the resolution (dpi) at which the PNG files are created.

\begin{verbatim}
options(tikzRasterResolution = 300)
\end{verbatim}

tikzPdfTeX WarnUTF A TRUE/FALSE value that controls whether warnings are printed if Unicode characters are sent to a device using the pdf\TeX{} engine.

For convenience the function setTikzDefaults() is provided which sets all the global options back to their original values.
The proper placement of a .Rprofile file is explained in ?Startup. For the details of why calling the \LaTeX compiler is necessary, see Part III.

### A Word of Caution About Setting Options.

A lot of power is given to you through these global options, and with great power comes great responsibility. For example, if you do not include the TikZ package in the tikzLatexPackages option then all of the string metric calculations will fail. Or if you use a different font when compiling than you used for calculating metrics, strings may be placed incorrectly. There are innumerable ways for packages to clash in \LaTeX so be aware.

## 4 The \texttt{tikz()} Function

### 4.1 Description

The \texttt{tikz()} function provides most of the functionality of the \texttt{tikzDevice} package. This function is responsible for creating new R graphics devices that translate the output of graphics functions to the TikZ format. The device supports many levels of output that range from stand-alone \LaTeX documents that may be compiled directly to code chunks that must be incorporated into existing \LaTeX documents using the \texttt{\include{}} function.

### 4.2 Usage

The \texttt{tikz()} function opens a new graphics device and may be called with the following arguments:

```r
\texttt{tikz(file = "Rplots.tex", width = 7, height = 7,}
\texttt{bg="transparent", fg="black", pointsize = 10, standAlone = FALSE,}
\texttt{bareBones = FALSE, console = FALSE, sanitize = FALSE,}
\texttt{engine = getOption("tikzDefaultEngine",}
\texttt{documentDeclaration = getOption("tikzDocumentDeclaration",}
\texttt{packages = getOption("tikzLatexPackages",}
\texttt{footer = getOption("tikzFooter"))}
```

- **file** A character string indicating the desired path to the output file. It is recommended, but not required, that the filename end in \texttt{.tex}.
- **width** The width of the output figure, in \texttt{inches}.
- **height** The height of the output figure, in \texttt{inches}.
- **bg** The starting background color for the plot.
- **fg** The starting foreground color for the plot.
- **pointsize** Base pointsize used in the \LaTeX document. This option is only referenced if a valid pointsize cannot be extracted from the value of \texttt{getOption("tikzDocumentDeclaration")}. See Subsection 4.3 for more details.
- **standAlone** A logical value indicating whether the resulting file should be suitable for direct processing by \LaTeX.
- **bareBones** A logical value indicating whether the resulting TikZ code produced without being placed within a \LaTeX tikzpicture environment.
console Should the output of tikz be directed to the R console (default FALSE). This is useful for dumping tikz output directly into a \LaTeX{} document via sink. If TRUE, the file argument is ignored. Setting file=" is equivalent to setting console=TRUE.

sanitize Should special latex characters be replaced (Default FALSE). See the section “Options That Affect Package Behavior” for which characters are replaced.

generate A string specifying which TeX engine to use. Possible values are 'pdftex' and 'xetex'.

documentDeclaration See Section 3.1, “Options That Affect Package Behavior.”

packages See Section 3.1, “Options That Affect Package Behavior.”

footer See Section 3.1, “Options That Affect Package Behavior.”

The first five options should be familiar to anyone who has used the default graphics devices shipped with R. The options file, width, height, bg and fg represent the standard graphics parameters currently implemented by tikzDevice. The last two options, standAlone and bareBones, are specific to the tikz() graphics device and affect the structure the output file. Using these options tikz() supports three modes of output:

- Graphics production as complete \LaTeX{} files suitable for compilation.
- Graphics production as complete figures suitable for inclusion in \LaTeX{} files.
- Graphics production as raw figure code suitable for inclusion in an enclosing \texttt{tikzpicture} environment in a \LaTeX{} file.

### 4.3 Font Size Calculations

The overarching goal of the ‘tikzDevice’ is to provide seamless integration between text in R graphics and the text of \LaTeX{} documents that contain those graphics. In order to achieve this integration the device must translate font sizes specified in R to corresponding font sizes in \LaTeX{}. The issue is that font sizes in \LaTeX{} are controlled by a “base font size” that is specified at the beginning of the document- typically 10pt. There is no easy way in \LaTeX{} to change the font size to a new numerical value, such as 16pt for a plot title. Fortunately, the TikZ graphics system allows text to be resized using a scaling factor. The tikzDevice calculates a scaling factor used to approximate other font sizes using the following three inputs:

- The “base font size” specified when the graphics device is created.
- The “character expansion factor” parameter, specified using the ‘cex’ argument to functions such as described in the documentation of the R function \texttt{par}().
- The “font size” parameter, specified using the ‘ps’ argument to functions such as \texttt{par}() or the ‘fontsize’ argument to functions such as \texttt{gpar}().

The calculation used is:

\[
\text{Scaling Factor} = \text{cex} \cdot \frac{\text{ps}}{\text{base font size}}
\]

The tricky bit is the specification of the “base font size”. By default the tikzDevice will attempt to determine this parameter by scanning the value of \texttt{options("tikzDocumentDeclaration")} using the regular expression \texttt{\\d+[pt]}. With the default header:

\begin{verbatim}
\documentclass[10pt]article
\end{verbatim}

This regular expression will return 10 as the base pointsize to be used by the device. If the regular expression fails to produce a match, the value of the \texttt{pointsize} argument to the \texttt{tikz()} function will be used.
4.4 UTF-8 Output

Version 0.6.0 introduced support for (multibyte) Unicode characters in the text of graphics. There are a few important implications/limitations of this feature:

**System Requirements:** A working version of \texttt{Xe\LaTeX} along with the packages \texttt{fontspec} and \texttt{xunicode} are required for direct processing of Unicode input. If \texttt{tikzDevice} cannot find \texttt{Xe\LaTeX} then Unicode support cannot be guaranteed.

**Encoding:** \texttt{tikzDevice} will try its best to convert characters from other encodings but do not count on it converting things correctly, best to do the conversion yourself beforehand to avoid unexpected output.

**Fonts:** Having a Unicode character actually show up in your \LaTeX document relies on the font you use having the glyph available. We leave it up to the user to know for themselves what is available. Otherwise you will likely just get no output in place of where the character should be.

**plotmath:** There is specifically no support for input of plotmath characters as unicode since the user can simply input \LaTeX math directly. We strongly encourage the use of \LaTeX math over plotmath for style and consistency’s sake. A consequence of this is that most of the \texttt{R} examples and demos of plotmath won’t work without significant manipulation (your mileage may vary but you may get anything from errors to warnings to documents that mysteriously won’t compile). That is not to say that the output could not be duplicated with \texttt{tikzDevice} but the examples will not work out of the box.

**Compiling:** A graphic that contains UTF-8 characters should be compiled with \texttt{Xe\LaTeX} (or possibly \texttt{Lua\LaTeX}) with the \texttt{xunicode} and \texttt{fontspec} packages enabled.

**ASCII only:** Everything should be exactly the same as previous versions if only ASCII (single byte) characters are used (i.e. character codes less than 132).

4.5 Examples

4.5.1 Default Mode

The most common use of the \texttt{tikz()} function is to produce a plot that will be included in another \LaTeX document, such as a report. Running the following example in \texttt{R} will produce a very simple graphic using the \texttt{plot()} function.

```r
require(tikzDevice)
tikz('simpleEx.tex', width=3.5, height=3.5)
plot(1,main='Hello World!')
dev.off()
```

A simple \LaTeX document is then required to display the output of the basic \texttt{tikz()} command. This document must include the TikZ as one of the packages that it loads. The TikZ package provides several optional libraries that provide additional functionality, however none of these libraries are currently required to use the output of \texttt{tikz()}. Inside the \LaTeX document, the contents of the file \texttt{simpleEx.tex} are imported using the \texttt{\include{}} command.
One of the most exciting aspects of the \texttt{tikz()} function is that it allows the inclusion of arbitrary \LaTeX\ code in plotting commands. An important issue to note is that many \LaTeX\ commands are prefixed by the backslash, \texttt{\textbackslash}, character. This character has a special meaning as an escape character in many computing applications, including \texttt{R}. Therefore, it is necessary to place two backslashes, \texttt{\textbackslash\textbackslash}, in the input to \texttt{R} commands in order to cause one to appear in the output. The next example demonstrates how to use \LaTeX\ commands in plot annotation.
require(tikzDevice)
tikz('latexEx.tex',
       width=3.5,height=3.5)
x <- rnorm(10)
y <- x + rnorm(5,sd=0.25)
model <- lm(y ~ x)
rsq <- summary(model)$r.squared
rsq <- signif(rsq,4)
plot(x,y,main='Hello \LaTeX !
')
abline(model,col='red')
mtext(paste("Linear model: $R^2=",
       rsq,"\$",line=0.5))
legend('bottomright', legend =
paste("$y = ",
       round(coef(model)[2],3), 'x +',
       round(coef(model)[1],3), '\$',
       sep=''), bty='n')
dev.off()

Figure 2: A more complicated example of tikz() usage incorporating natively rendered \LaTeX commands.

4.5.2 bareBones Mode

bareBones output is designed to facilitate inclusion of code generated by tikz() into a larger TikZ graphic. Normally tikz() packages the commands it produces as a self-contained figure. This is done by placing the \begin{tikzpicture} and \end{tikzpicture} commands at the beginning and end of the output file. When bareBones is invoked, the tikzpicture environment is omitted which allows the output to be embedded inside another tikzpicture of the users own construction.

require(tikzDevice)
require(maps)
tikz('westCoast.tex', bareBones=TRUE)
map('state', regions=c('california', 'oregon', 'washington'),
     lwd=4, col='grey40')
eurekaLon <- grconvertX(-124.161, to='device')
eurekaLat <- grconvertY(40.786, to='device')
longviewLon <- grconvertX(-122.962, to='device')
longviewLat <- grconvertY(46.148, to='device')
coosLon <- grconvertX(-124.237, to='device')
coosLat <- grconvertY(43.378, to='device')
sfLon <- grconvertX(-122.419, to='device')
sfLat <- grconvertY(37.775, to='device')
tikzAnnotate(paste('\coordinate (humBay) at (',
                  eurekaLon, ',', eurekaLat, ');', sep=''))
tikzAnnotate(paste('\coordinate (longView) at (',
                  longviewLon, ',', longviewLat, ');', sep=''))
The `\include{}` command may now be used to import the device output into another `tikzpicture`. The included code must be wrapped in a `scope` environment that contains the options `x=1pt` and `y=1pt`. This informs TikZ of the units being used in the coordinates of the plot output. The options `xshift` and `yshift` may also be applied to the `scope` in order to position the plot. The following code demonstrates how to embed `bareBones` output in a `tikzpicture`:

```
\begin{tikzpicture}
\begin{scope}[x=1pt,y=1pt]
\input{figs/westCoast}
\end{scope}
\foreach \name / \port in {
  Longview / longView,
  Coos Bay / coosBay,
  Humboldt Bay / humBay,
  Oakland / sfBay
} {
  \node[circle,draw,ultra thick,fill=green!60!brown!40,outer sep=6pt,minimum size=12pt,pin={[draw,ultra thick,rounded corners,pin edge={black,ultra thick,<-,,>=stealth}]180:\name}] at (\port) {};
}
\end{tikzpicture}
```
4.5.3 standAlone Mode

When the standAlone option is passed to tikz(), the resulting .tex file will be a complete \LaTeX{} document designed to be compiled on its own. This means that in addition to \begin{tikzpicture} and \end{tikzpicture} the file will also contain \begin{document}, \end{document} and a \LaTeX{} preamble. The preview package is also used in files produced by standAlone and is used to crop the pages in the resulting document to the bounding boxes of the figures that it contains. Stand-alone output may be produced in the following manner:
Note that files produced using the \texttt{standAlone} option should not be included in \LaTeX\ documents using the \texttt{\input{}} command! Use \texttt{\includegraphics{}} or load the \texttt{pdfpages} package and use \texttt{\includepdf{}}.

4.5.4 console output Mode

Version 0.5.0 of \texttt{tikzDevice} introduced the \texttt{console} option. Instead of sending its output to a file \texttt{tikz} will send it output to \texttt{stdout}. This kind of output can be redirected to a file with \texttt{sink()} or spit out directly into a \TeX\ document from a \texttt{Sweave} file so that the \TeX\ file is self contained.

4.5.5 Getting around special \LaTeX\ characters

4.5.6 Using the Global Options: An Xe\LaTeX\ example

It is also possible to use other typesetting engines like Xe\LaTeX\ by using the global options provided by \texttt{tikzDevice}. The following example was inspired by Dario Taraborelli and his article The Beauty of \LaTeX.\TeX.
# Set options for using XeLaTeX

```r
options(tikzLatex = 'xelatex')
options(tikzDocumentDeclaration = '\documentclass { article }')
```

# The preview package must be loaded first with the xetex driver option

```r
options( tikzLatexPackages = c("\usepackage[active,tightpage,xetex]{preview}\PreviewEnvironment{pgfpicture}\setlength{\PreviewBorder}{0pt}\usepackage[fontspec]\usepackage{colorlinks, breaklinks, pdftitle={The Beauty of LaTeX},\pdfauthor={Taraborelli, Dario}]\{hyperref\}\usepackage{tikz}\definecolor{Gray}{rgb}{.7,.7,.7}\definecolor{lightblue}{rgb}{.2,.5,1}\definecolor{myred}{rgb}{1,0,0}\newcommand{\red}[1]{\color{myred} #1}\newcommand{\reda}[1]{\color{myred}\fontspec\[Variant =2\]{Zapfino}#1}\newcommand{\redb}[1]{\color{myred}\fontspec\[ Variant =3\]{Zapfino}#1}\newcommand{\redf}[1]{\color{myred}\fontspec\[ Variant =7\]{Zapfino}#1}\newcommand{\lbl}[1]{\color{lightblue} #1}\newcommand{\lbla}[1]{\color{lightblue}\fontspec\[Variant =2\]{Zapfino}#1}\newcommand{\lbfb}[1]{\color{lightblue}\fontspec\[Variant =7\]{Zapfino}#1}\newcommand{\old}[1]{\fontspec\[Ligatures ={ Common , Rare }\,\Swashes ={LineInitial , LineFinal}\]{Zapfino}\fontsize{25 pt}{3/zero.noslash pt}\selectfont #1}\newcommand{\smallprint}[1]{\fontspec{Hoefler Text}\fontsize{13 pt}{13 pt}\color{Gray}\selectfont #1}}
```

# Set the content using custom defined commands

```r
label <- c("\noindent {\red d} roo {\lbl g}\" "\noindent {\reda d} roo {\lbla g}\" "\noindent {\redb d} roo {\lbfb g}\\[.3 cm]\" "\noindent {\redf d} roo {\lbfb g}\"")
```

Xe\LaTeX\ Example
Compiling the resulting file `xelatexEx.tex` like so:

```
xelatex xelatexEx.tex
```

will produce the output in Figure 4! Please note some of the fonts used in the example may not be available on your system.
4.5.7 Annotating Graphics with TikZ Commands

The function `tikzAnnotate` provides the ability to annotate your graphics with TikZ commands. There are a lot of exciting possibilities with this feature; it basically opens up the door for you to draw anything on your plot that can be drawn with TikZ. Check out the results in Figure 5.

```r
require(tikzDevice)
# Load some additional TikZ libraries
tikz("annotation.tex", width=4, height=4, packages = c(getOption('tikzLatexPackages'), "\usetikzlibrary{decorations.pathreplacing}", "\usetikzlibrary{positioning}", "\usetikzlibrary{shapes.arrows, shapes.symbols}"
)
p <- rgamma(3/zero.noslash/zero.noslash, 1)
outliers <- which(p > quantile(p,.75)+1.5*IQR(p))
boxplot(p)
# Add named coordinates that other TikZ commands can hook onto
```

Figure 4: Result of Xe\LaTeX{} example
tikzCoord(1, min(p[outliers]), 'min outlier')
tikzCoord(1, max(p[outliers]), 'max outlier')
# Use tikzAnnotate to insert arbitrary code, such as drawing a
# fancy path between min outlier and max outlier.
tikzAnnotate(c("\draw[very thick,red,",
    # Turn the path into a brace.
    'decorate,decoration={brace,amplitude=12pt},',
    # Shift it 1em to the left of the coordinates
    'transform canvas={(xshift=-1em)}',
    '(min outlier) --',
    # Add a node with some text in the middle of the path
    'node [single arrow,anchor=tip,fill=white,draw=green,",
    'left=14pt,text width=0.70in,align=center]',
    '(Holy Outliers Batman!)', '(max outlier);'))
# tikzNode can be used to place nodes with customized options and content
  tikzNode(
    opts='starburst,fill=green,draw=blue,very thick,right=of max outlier',
    content='Wow!'
  )
dev.off()
4.5.8 tikz() vs. pdf() for plotmath symbols and Unicode characters

This is a side-by-side example showing how tikz(..., engine = ‘xetex’) handles UTF-8 characters and plotmath symbols compared to the standard R pdf() device.

Figure 6: example(text) using the standard pdf() device.

Figure 7: example(text) using tikz(..., engine = ‘xetex’).
5  The getLatexCharMetrics() and getLatexStrWidth() Functions

5.1 Description

These two functions may be used to retrieve font metrics through the interface provided by the \texttt{tikzDevice} package. Cached values of the metrics are returned if they have been calculated by the \texttt{tikzDevice} before. If no cached values exist, the \LaTeX{} compiler will be invoked to generate them.

5.2 Usage

The font metric functions are called as follows:

\begin{verbatim}
getLatexStrWidth( texString, cex = 1, face = 1)  
getLatexCharMetrics( charCode, cex = 1, face = 1)
\end{verbatim}

\begin{itemize}
  \item \texttt{texString} A string for which to compute the width. \LaTeX{} commands may be used in the string, however all backslashes will need to be doubled.
  \item \texttt{charCode} An integer between 32 and 126 which indicates a printable character in the ASCII symbol table using the T1 font encoding.
  \item \texttt{cex} The character expansion factor to be used when determining metrics.
  \item \texttt{face} An integer specifying the R font face to use during metric calculations. The accepted values are as follows:
    \begin{enumerate}
      \item 1: Text should be set in normal font face.
      \item 2: Text should be set in \textbf{bold font face}.
      \item 3: Text should be set in \textit{italic font face}.
      \item 4: Text should be set in \textbf{bold italic font face}.
      \item 5: Text should be interpreted as \texttt{plotmath} symbol characters. Requests for font face 5 are currently ignored.
    \end{enumerate}
\end{itemize}

5.3 Examples

The \texttt{getLatexStrWidth()} function may be used to calculate the width of strings containing fairly arbitrary \LaTeX{} commands. For example, consider the following calculations:

\begin{verbatim}
getLatexStrWidth("The symbol: alpha")
[1] 82.5354
getLatexStrWidth("The symbol: \$\alpha\$")
[1] 65.08636
\end{verbatim}

For the first calculation, the word “alpha” was interpreted as just a word and the widths of the characters ‘a’, ‘l’, ‘p’, ‘h’ and ‘a’ were included in the string width. For the second string, $\alpha$ was interpreted as a mathematical symbol and only the width of the symbol ‘α’ was included in the string width.

The \texttt{getLatexCharWidth()} function must be passed an integer corresponding to an ASCII character code and returns three values:
• The ascent of the character - the distance between the baseline and the highest point of the character’s glyph.
• The descent of the character - the distance between the baseline and the lowest point of the character’s glyph.
• The width of the character.

The character ‘y’ has an ASCII symbol code of 121 and possesses a tail that descends below the text line. Therefore a non-zero value will be returned for the descent of ‘y’. The character ‘x’, ASCII code 120, has no descenders, so its descent will be returned as zero.

```r
# Get metrics for 'y'
getLatexCharMetrics(121)
[1] 4.30450 1.94397 5.27649

# Get metrics for 'x' - the second value is the descent
# and should be zero or very close to zero.
getLatexCharMetrics(120)
[1] 4.30450 0.00000 5.27649
```

Note that characters, along with numbers outside the range of [32-126], may not be passed to the `getLatexCharMetrics()` function. If for some reason a floating point number is passed, it will be floored through conversion by `as.integer()`.

```r
getLatexCharMetrics('y')
NULL
getLatexCharMetrics(20)
NULL

# Will return metrics for 'y'
getLatexCharMetrics(121.99)
[1] 4.30450 1.94397 5.27649
```
Part II

Installation Guide

This section is intended to offer pointers on how to obtain a \LaTeX{} distribution if there is not one already installed on your system. The distributions detailed in this section are favorites of the tikzDevice developers as they integrated package managers which greatly simplify the process of installing additional \LaTeX{} packages. Currently this section is not, and may never be, a troubleshooting guide for \LaTeX{} installation. For those unfortunate situations we refer the user to the documentation of each distribution.

6 Obtaining a \LaTeX{} Distribution

A \LaTeX{} distribution provides the packages and support programs required by the tikzDevice and the documents that use its output. In addition a \LaTeX{} compiler, a few extension packaged are required. Section 7 describes how to obtain and install these packages.

6.1 Windows

Windows users will probably prefer the MiKTeX distribution available at http://www.miktex.org. An amazing feature of the MiKTeX distribution is that it contains a package manager that will attempt to install missing packages on-the-fly. Normally when \LaTeX{} is compiling a document that tries to load a missing package it will wipe out with a warning message. When the MiKTeX compilers are used compilation will be suspended while the new package is downloaded.

6.2 UNIX/Linux

For users running a Linux or UNIX operating system, we recommend the TeX Live distribution which is available at http://www.tug.org/texlive/acquire.html. TeX Live is maintained by the TeX Users Group and a new version is released every year. Note that the version of TeX Live provided by many Linux package management systems is the 2007 version. We recommend using TeX Live 2008 or higher as the \texttt{tlmgr} package manager was introduced in the 2008 distribution. Using \texttt{tlmgr} greatly simplifies the adding and removing packages from the distribution. The website offers an installation package, called \texttt{install-tl.tar.gz} or something similar, that contains a shell script that can be used to install an up-to-date version of the TeX Live distribution.

6.3 Mac OS X

For users running Apple’s OS X, we recommend the Mac TeX package available at http://www.tug.org/mactex/. Mac TeX is basically TeX Live packaged inside a convenient OS X installer along with a few add-on packages. One striking difference between the Mac TeX and TeX Live installers is that the installer for Mac TeX includes the whole TeX Live distribution in the initial download- for TeX Live 2008 this amounts to approximately 1.2 GB. This is quite a large download that contains several packages that the average or even advanced user will never ever use. To conserve time and space we recommend installing from the basic installer at http://www.tug.org/mactex/morepackages.html and using the \texttt{tlmgr} utility to add desired add-on packages.

Adam R. Maxwell has created a very nice graphical interface to \texttt{tlmgr} for OS X called the TeX Live Utility. It may be obtained from http://code.google.com/p/mactlmgr/ and we highly recommend it.
7 Installing TikZ and Other Packages

Unsurprisingly, \texttt{tikzDevice} requires the TikZ package to be installed and available in order to function properly. TikZ is an abstraction of a lower-level graphics language called \texttt{PGF} and both are distributed as the the \texttt{pgf} package.

7.1 Using a \LaTeX Package Manager

The easiest way to install \LaTeX packages is by using a distribution that includes a package manager such as MiKTeX or TeX Live/Mac TeX. For Windows users, the MiKTeX package manager usually handles package installation automagically during compilation of a document that is requesting a missing package. The MiKTeX package manager, \texttt{mpm}, can also be run manually from the command prompt:

\begin{verbatim}
mpm --install packagename
\end{verbatim}

For versions of TeX Live and Mac TeX dated 2008 or newer, the \texttt{tlnmr} package manager is used in an almost identical manner:

\begin{verbatim}
tlnmr install packagename
\end{verbatim}

7.2 Manual Installation

Sometimes an automated package manager cannot be used. Common reasons may be that one is not available, as is the case with the TeX Live 2007 distribution, or that when running the package manager you do not have write access to the location where \LaTeX packages are stored, as is the case with accounts on shared computers. If this is the case, a manual install may be the best option for making a \LaTeX package available.

Generally, the best place to find \LaTeX packages is the Comprehensive TeX Archive Network, or CTAN located at \url{http://www.ctan.org}. In the case of the PGF/TikZ package, the project homepage at \url{http://www.sourceforge.net/projects/pgf} is also a good place to obtain the package—especially if you would like to play with the bleeding-edge development version.

Generally speaking, all \LaTeX packages are stored in a specially directory called a \texttt{texmf} folder. Most \TeX distributions allow for each user to have their own personal \texttt{texmf} folder somewhere in their home path. The most usual locations, and here \texttt{usual} is and unfortunately loose term, are as follows:

\begin{verbatim}
For UNIX/Linux

\texttt{/texmf}
\end{verbatim}
For Mac OS X

~/.Library/texmf

For Windows, using MiKTeX

# None predefined. However the following command will open
# the MiKTeX options panel and a new texmf folder may be assigned
# under the "Roots" tab.

The location of files and subfolders in the texmf directory should follow a standard pattern called the TeX Directory Structure or TDS which is documented here: [http://tug.org/tds/tds.pdf](http://tug.org/tds/tds.pdf). Fortunately, most packages available on CTAN are archived in such a way that they will unpack into a TDS-compliant configuration. TDS-compliant archives usually have the phrase tds somewhere in their filename and may be installed from a UNIX shell¹ like so:

**Installing \LaTeX{} package archives**

# For zip files.
unzip package.tds.zip -d /path/to/texmf

# For tarballs.
tar -xzf -C /path/to/texmf package.tar.gz

For packages that aren’t provided in TDS-compliant form look for installation notes—usually provided in the form of an INSTALL file. If all else fails \LaTeX{} packages can usually be installed by copying the files ending in .sty to texmf/tex/latex/.

After package files have been unpacked to a texmf folder, the database of installed packages needs to be updated for the \LaTeX{} compiler to take notice of the additions. This is done with the mktexlsr command:

**Registering new \LaTeX{} packages**

mktexlsr

# Successful package installation can be checked by running the
# kpsewhich command. For a package accessed in a document
# by \usepackage{package}, kpsewhich should return a path to
# package.sty
kpsewhich tikz.sty
/Users/Smithe/Library/texmf/tex/latex/pgf/frontendlayer/tikz.sty

---

¹Sorry Windows users, we enjoy using command prompt about as much as a poke in the eye with a sharp stick. Hence we don’t use it enough to offer advice. May we suggest Cygwin?
Part III

Package Internals

We will encourage you to develop the three great virtues of a programmer: laziness, impatience, and hubris.

Programming Perl
–LARRY WALL

8 Introduction and Background

<table>
<thead>
<tr>
<th>Caveat Lector</th>
</tr>
</thead>
<tbody>
<tr>
<td>The following introduction currently presents a vision(delusion?) of what this section of the documentation could be. Currently the only portion of the inner workings of this package that we attempt to explain and document are those related to using the \LaTeX{} compiler to obtain font metrics.</td>
</tr>
</tbody>
</table>

We learn best through working with examples. When it comes to programming languages this involves taking working code that someone else has written, breaking it in as many places at it can possibly be broken, and then trying to build something out of the wreckage. Open source software facilitates this process wonderfully by ensuring the source code of a project is always available for inspection and experimentation. The tikzDevice itself was created by disassembling and then rebuilding Valerio Aimale’s PicTeX device driver which is a part of the R core codebase.

This section is our attempt to help anyone who may be experimenting with our code, and by extension the internals of the R graphics system. There may also be useful, or useless, tidbits concerning building R packages and interacting with the core R language. The R language can be extended in so many interesting and useful ways and it is our hope that the following documentation may provide a case study for anyone attempting such an extension.

We will make an attempt to assume no special expertise with any of the systems or programming languages leveraged by this package and described by this documentation. Therefore, if you are an experienced developer and find yourself thinking “My god, are they really about to launch into a description of how C header files work?”, please feel free to skip ahead a few paragraphs. We received our formal introduction to computer programming in a college engineering program- therefore our programming background is rooted in Fortran (or, if you prefer, FORTRAN). We are attempting to write the sort of documentation that we would have found invaluable at the start of this project.

Therefore, this section is for all the budding developers like ourselves out there- people who have done some programming and who are starting to take a close look at the nuts and bolts of the R programming environment. If you feel like you are wandering through a vast forest getting smacked in the face by every branch then maybe this section will help pull some of those branches out of the way...

...then again we have a lot of material to cover: R, C, \LaTeX{}, TikZ, typography and the details of computerized font systems. Our grip may fail and send those branches flying back with increased velocity.

We wish you luck!

–The tikzDevice Team
9 Anatomy of an R Graphics Device

The core of an R graphics device is a collection of functions, written in C, that perform various specialized tasks. A description of some of these functions can be found in the R Internals manual while the main documentation is in the C header file GraphicsDevice.h. For most R installations this header file can be found in the directory R_HOME/include/R_ext. For copies of R distributed in source code form, GraphicsDevice.h is located inside R-version/src/include/R_ext. The following is a description of the functions each graphics device is expected to provide:

**Drawing Routines**

circle This function is required to draw a circle centered at a given location with a given radius.

clip This function specifies a rectangular area to be used as a clipping boundary for any device output that follows.

draw This function draws a line between two points.

polygon This function draws lines between a list of points and then connects the first point to the last point.

polyline This function draws lines between a list of points.

rect This function is given a lower left corner and an upper right corner and draws a rectangle between the two.

text This function inserts text at a given location.

**Font Metric Routines**

metricInfo This function is given the name of a single character and reports the ascent, descent and width of that character.

strWidth This function is given a text string and reports the width of that string.

**Utility Routines**

activate This function is called when the device is designated as the active output device - i.e. by using dev.set() in R.

close This function is called when the device is shut down - i.e. by using dev.off() in R.

deactivate This function is called when another device is designated as the active output device.

locator This function is mainly used by devices with a GUI window and reports the location of a mouseclick.

mode This function is called when a device begins drawing output and again when the device finishes drawing output.

newPage This function initiates the creation of a new page of output.

size This function reports the size of the canvas the device is drawing on.

10 Calculating Font Metrics

Font metrics are measurements associated with the glyphs, or printed characters, of a particular font. R requires three of these metrics in order to produce correctly aligned output. The three metrics graphics devices are required to supply are:

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ascent</td>
<td>Ascent is the distance between the baseline and the tallest point on a character’s glyph. For the ‘g’ printed to the left, the ascent has been calculated as: 24.99963pt</td>
</tr>
<tr>
<td>Descent</td>
<td>Descent is the distance between the baseline and the lowest point on a character’s glyph. For the ‘g’ printed to the left, the descent has been calculated as: 7.77771pt</td>
</tr>
</tbody>
</table>
Providing font metrics and string widths is without a doubt most difficult task a R graphics device must undertake. The calculation of string widths is made even difficult for the \texttt{tikzDevice} as we attempt to process arbitrary \LaTeX\ strings. Inside R the string “$\alpha$” literally has 8 characters, but when it is typeset it only has one: $\alpha$.

Calculating font metrics is a tricky business to begin with and the fact that the typeset representation of a \LaTeX\ string is different from its representation in source code compounds the difficulty of the task immensely. Therefore, we took the path of laziness and started looking for an easy way out (remember the three great virtues of a programmer?). The solution we came up with seemed easy enough– make \LaTeX\ calculate these metrics for us, after all that is what a \LaTeX\ compiler does for a living.

Now, how to do that?

10.1 Character Metrics

As a starting point, let’s examine the interface of the C function that R calls in order to determine character metrics:

```c
void (metricInfo)(int c, const pGEcontext gc ,
     double* ascent, double* descent, double* width,
     pDevDesc dd);
```

The most important variables involved in the function are \texttt{c}, \texttt{ascent}, \texttt{descent} and \texttt{width}. The incoming variable is \texttt{c}, which contains the character for which R is requesting font metrics. Interestingly, \texttt{c} is passed as an integer, not a character as one might expect. What’s up with that? Well, the short answer is that R passes the ASCII or UTF8 symbol code of a character and not the character it’s self. How to use that character code to recover a character will be explained later.

The outgoing variables are \texttt{ascent}, \texttt{descent} and \texttt{width}. The asterisks, \texttt{*}, in their definitions mean these variables are passed as pointers as opposed to values. A complete discussion of the differences between pointers and values could, and has, filled up several chapters of several programming books. The important distinction in context of the \texttt{metricInfo} function is that when a number is assigned to a pointer variable, that number is available elsewhere after the function terminates. In contrast, when a number is assigned to a value variable, that number disappears when the function ends unless it is explicitly sent back out to the wide world through the \texttt{return} statement. So, the main task of the \texttt{metricInfo} function is to assign values to \texttt{ascent}, \texttt{descent} and \texttt{width}.

The other two variables present in the function are the \texttt{pGEcontext} variable \texttt{gc} and the \texttt{pDevDesc} variable \texttt{dd}. \texttt{gc} contains information such as the font face, foreground color, background color, character expansion factor, etc. currently in use by the graphics system. \texttt{dd} is the object which contains R’s representation of the graphics device. For the sake of simplifying the following discussion, we will ignore these variables.

So, to recap– we have an integer \texttt{c} coming in that represents a code for a character in the ASCII or UTF8 symbol tables (for the sake of the following discussion, we will assume ASCII characters only). Our overall task is to somehow turn that integer into three numbers which can be assigned to the pointer variables \texttt{ascent}, \texttt{descent} and \texttt{width}. And, since we’re being lazy, we’ve decided that the best way to do that is to ask the \LaTeX\ compiler to compute the numbers for us.

Recovering these numbers from the \LaTeX\ compiler involves the execution of three additional tasks:
1. We must write a \LaTeX{} input file that contains instructions for calculating the metrics.

2. We call the \LaTeX{} compiler to process that input file.

3. We must read the compiler’s output in order to recover the metrics.

Each of these tasks could be executed from inside our C function, \texttt{metricInfo}. However, we will run into some difficulties—namely with step 2, which involves calling out to the operating system with orders to run \LaTeX{}. Each operating system handles these calls a little differently and our package must attempt to get this job done whether it is running on Windows, UNIX, Linux or Mac OS X.

Portable C code could be written to handle each of these situations, but that is starting to sound like work and we’re trying to be lazy here. What we need is to be able to work at a higher \textit{level of abstraction}. That is—instead of using C, we need to be working inside a language that shields us from such details as what operating system is being used. R may have called this C function to calculate font metrics, but we really want to do the actual computations back inside R.

### 10.2 Calling R Functions from C Functions

The “Ritual of the Calling of the R Function” is easy enough to perform as long as you don’t have burning need to know all the details of the objects you are handling. The C level representation of a R object such as a variable or function is an object known as a \texttt{SEXP}. For the exact details on what a \texttt{SEXP} is and how it works, we refer the interested user to chapter one of the \textit{R Internals} manual.

The R function we will be calling is declared in the R environment as follows:

```
\begin{verbatim}
getLatexCharMetrics <- function( charCode ){
    # System call to LaTeX
}
\end{verbatim}
```

In order to call this function for C, we need a vector composed of two C-level R objects— one containing the name of the function we are calling and another one containing the value we are passing for \texttt{charCode}. This is set up in C as follows:

```
\begin{verbatim}
void ( metricInfo )( int c, const pGEcontext gc , double* ascent , double* descent , double* width , pDevDesc dd ){
    SEXP RCallBack;
    PROTECT( RCallBack = allocVector( LANGSXP , 2) );
    SEXP metricFun = findFun ( install("getLatexCharMetrics"), R_Global_Env );
    SETCAR( RCallBack , metricFun );
    SETCADR( RCallBack , ScalarInteger( c ) );
}
\end{verbatim}
```
The first thing that happens in the code chunk above is that a new SEXP variable named RCallBack is created. This variable will be the agent through which we will communicate with the R environment. The next action is to allocate our callback variable as a vector of length 2— we need one slot for the R function name and one slot for the value that is being passed into the function. This allocation happens inside the R environment, so it is executed inside the PROTECT statement. The reason for using PROTECT is that the R garbage collector is constantly on the prowl for unused objects in the R environment. An object is considered “unused” if it is not attached to any variable name in the R environment. Since the object is only attached to the variable RCallBack in our C function, the R garbage collector will see it a valid candidate for deletion. The purpose of PROTECT is to keep our new vector from being trashed.

The next portion of the C function retrieves the R function object for getLatexCharMetrics. The function is searched for in R global namespace, so it must be one that is available to the user from the R command prompt when the package is loaded. The function is stored in the SEXP variable metricFun. We do not have to involve PROTECT in the assignment since getLatexCharMetrics exists as a variable name in the R environment.

The last portion of the code chunk is responsible for loading the function name and call value into RCallBack. The CAR statement is used to retrieve the value of a SEXP variable and the SETCAR statement is used to set the value of a SEXP. In this case we use SETCAR to designate the R function stored in metricFun as the first value of RCallBack.

When dealing with a vector SEXP such as RCallBack, which has 2 slots, we need to use a different function to access the second slot. The CDR function will allow us to move to the second slot in RCallBack where we may perform a SETCAR to specify a value. In the example code, these operations were combined by using the SETCADR function which has the same effect as:

```
SETCADR( CDR( RCallBack ), ScalarInteger( c ) );
```

After assigning the value of the C variable c as the second value of RCallBack, we need to 'tag' it as the value that corresponds to the charCode argument of getLatexCharMetrics. This is done by using the SET_TAG function. Once again, we use CDR to shift our area of operation to the second slot of RCallBack. Now that the RCallBack object is set up, we are ready to actually call the getLatexCharMetrics function.

```
SEXP LatexMetrics;
PROTECT( LatexMetrics = eval( RCallBack, R_GlobalEnv ) );
```

And that’s it! We create a new SEXP to hold the return values of getLatexCharMetrics and execute the eval function to cause getLatexCharMetrics to be executed inside the R environment. The details of the R function will be explained in the next section, for now let’s assume that it returns the three values we’re interested in as a vector of three numbers. How do we extract these values and assign then to ascent, descent and width?
Here the \texttt{REAL} function is used to coerce the \texttt{SEXP} variable \texttt{RMetrics} to a vector of real numbers. These numbers are then extracted and assigned to the return values of \texttt{metricInfo}. In C we must specify the ‘first’ value in a vector using the index 0 rather than the index 1 \footnote{There are good logical reasons for this from the point of view of a computer scientist-- but if your background in arrays is rooted in linear algebra it will be a bit disorienting.}. The last thing to do is release the restrictions we placed on the \texttt{R} garbage collector. Since we used the \texttt{PROTECT} function twice, we must call \texttt{UNPROTECT} and pass 2 as the argument.

### 10.3 Implementing a System Call to \LaTeX

Now we may turn to the actual guts of the \texttt{R} function \texttt{getLatexCharMetrics}. The first thing we need to do is set up a file for \LaTeX input:

```r
getLatexCharMetrics <- function ( charCode ){
  texDir <- tempdir()
  texLog <- file.path( texDir , 'tikzStringWidthCalc.log' )
  texFile <- file.path( texDir , 'tikzStringWidthCalc.tex' )
  texIn <- file( texFile, 'w')
  # To be continued...
}
```

The first thing we do is choose a place to create this input file. Now, when the \LaTeX compiler is run on a .tex file, a lot of additional files get created-- the whole process is a bit messy. Since the user probably wouldn’t appreciate having to clean up our mess, we use the \texttt{tempdir()} function to retrieve a path to a temporary directory on the system. Here is the first place we benefit from the added level of abstraction granted by \texttt{R}. Each operating system has different locations for temporary directories. If we were still working in C, we would have to worry about such details. \texttt{R} takes care of those details for us.

Now that we have a place to work, we set up a couple of filenames- one for the input file, which ends in .tex and one for the \LaTeX log file, which ends in .log. We then open the .tex file for writing. The next step is to setup the preamble of the \LaTeX file.
Here we have started a standard \LaTeX input file by specifying `article` as the document class. We also add the `fontenc` package and specify `T1` as its option. This ensures we are using the Type 1 font encoding—by default \TeX and \LaTeX use an encoding called `OT1`. Why do we need to worry about font encodings? Well, a font encoding specifies which ASCII symbol codes map to which characters and by default, \TeX expects us to be using the Type 1 encoding (\TeX does support other encodings—but we’re ignoring that for now). For example, in the Type 1 encoding, the character that corresponds to the ASCII code 60 is the less-than sign: `<'. If we were to allow \TeX to retain its default `OT1` encoding, that same character code would instead map to an upside-down exclamation point: `¡'.

The other two packages we load are the `tikz` package and its `calc` library. Essentially we will have \TiKZ drop the character into a box and report some measurements concerning the size of that box. The last command, `batchmode` tells \LaTeX that there isn’t any user available to interact with—so it should bother to stop and ask any questions while processing this file.

The next step is to set up the part of the \LaTeX file that will actually calculate and report the widths we are looking for. As mentioned before, this is done by setting the character inside a \TiKZ node and extracting the dimensions of the box that surrounds it. In an attempt to improve clarity, the following code will be presented as straight \LaTeX—`getLatexCharMetrics` inserts it into the `texIn` file by means of `writeLines` as we have been doing all along. The string highlighted in red should be replaced with the value of the variable `charCode` that was passed in to the function `getLatexCharMetrics`.

What the heck just happened? Well, first we instructed \LaTeX to enter the \TiKZ picture environment using `\begin{tikzpicture}`. Then we ordered \TiKZ to create a node named `char` containing the command `\char{charCode}`.
followed by the value of charCode. For example, if we were passed ‘103’ as the character code, which corresponds to the character ‘g’, the node line should be:

\node[inner sep=0pt, outer sep=0pt] (char) {\char103};

The \texttt{inner sep} and \texttt{outer sep} options are set to \texttt{0pt} in order to ensure the boundaries of the node ‘hug’ the contents tightly. Now the whole point of setting the character inside a node is that Ti\kZ defines ‘anchors’ along the bounding box of the node. All anchors are referred using a \texttt{node name.position} notation. Since we named the node \texttt{char}, all the anchors start with \texttt{char}. The anchor positions relevant to our problem are shown below:

![Node Bounding Box](Image)

The ‘base’ anchor sits on the baseline of the text—therefore to calculate the ascent of the character ‘g’, all we have to do is figure out the difference in height between the positions \texttt{char.north} and \texttt{char.base}. Similarly, for the descent we would calculate the difference in height between \texttt{char.base} and \texttt{char.south} and width can be obtained using \texttt{char.west} and \texttt{char.east}. This is the purpose of the admittedly cryptic \texttt{\path} commands that are inserted in the \LaTeX{} input file. Let’s examine one of them:

\begin{verbatim}
\path let \p1 = ($(char.north) - (char.base)$), \n1 = {veclen(\x1,\y1)} in node{\typeout{tikzTeXAscent=\n1} }
\end{verbatim}

So, what exactly is going on here? Normally, the \texttt{\path} command is used to draw lines between points and add additional coordinates or nodes along those lines. For example, the command:

\begin{verbatim}
\path[draw] (0,0) -- (1,1) node {Hi!};
\end{verbatim}

Draws a line from (0,0) to (1,1) and places a node at (1,1) containing the word ‘Hi!’ In the Ti\kZ code produced by \texttt{getLatexCharMetrics}, the \texttt{let} operation is specified. Basically, \texttt{let} postpones the actual drawing of a \texttt{path} and performs calculations until the \texttt{in} keyword is encountered. The result of these calculations are stored in a set of special variables which must start with \texttt{\n}, \texttt{\p}, \texttt{\x} or \texttt{\y}. The first \texttt{let} operation executed is:
This performs a vector subtraction between the coordinates of \texttt{char.north} and \texttt{char.base}. The resulting x and y components are stored in the 'point' variable \( \mathbf{p}_1 \). The second operation executed is:

\[
\mathbf{n}_1 = \{\text{veclen}(\mathbf{x}_1, \mathbf{y}_1)\}
\]

This code \texttt{let} operation treats the coordinates stored in \( \mathbf{p}_1 \) as a vector and calculates its magnitude. The ‘1’ appended to the \texttt{x} and \texttt{y} variables specifies that we are accessing the x and y components of \( \mathbf{p}_1 \). This result is stored in the ‘number’ variable \( \mathbf{n}_1 \). Now, that our metric is stored in \( \mathbf{n}_1 \), our final task is to ensure it makes it into the \LaTeX\ .log file—this is done by adding a node containing the \texttt{typeout} command. The contents of the node:

\[
\texttt{typeout\{tikzTexAscent=\n}_1\}
\]

Cause the phrase ‘tikzTexAscent=’ to appear in the .log file—followed by the ascent calculated using the node anchors. After the ascent, descent and width have been calculated the \LaTeX\ compiler may be shut down, this is done by adding the final two lines to the input file:

```latex
\makeatother \\
\@@end
```

Now that the input file has been prepped, we must process it using the \LaTeX\ compiler and load the contents of the resulting .log so that we may search for the metrics we dumped using \texttt{typeout}.

```r
latexCmd <- getOption('tikzLatex')
llatexCmd <- paste(latexCmd, '-interaction=batchmode', 
                   '-output-directory', texDir, texFile)
silence <- system(latexCmd, intern=T, ignore.stderr=T)
texOut <- file(texLog, 'r')
logContents <- readLines(texOut)
close(texOut)
```

Terminating a \LaTeX\ compilation

The \LaTeX\ compiler is executed through the \texttt{system} function which handles the details of implementing a system call on whatever operating system we happen to be using. We assign the return value of the \texttt{system}
function to a dummy variable called \texttt{silence} so that no output floods the user’s screen. The last task is to extract our metrics from the text of the log we loaded.

```r
match <- logContents[ grep("tikzTeXWidth='", logContents) ]
width <- gsub('[A-Za-z]', '', match)

match <- logContents[ grep("tikzTeXAscent='", logContents) ]
ascent <- gsub('[A-Za-z]', '', match)

match <- logContents[ grep("tikzTeXDescent='", logContents) ]
descent <- gsub('[A-Za-z]', '', match)

return( as.double( c(ascent, descent, width) ) )
```

Here we use the \texttt{grep} function to search through the log output for the tags \texttt{tikzTeXWidth=}’, \texttt{tikzTeXAscent=} and \texttt{tikzTeXDescent=} that we specified when we used \texttt{\typeout}. After we recover a line containing one of these tags, we use the \texttt{gsub} command to remove the letters and the equals sign from the text line—leaving just the number we’re interested in. These values are then coerced using \texttt{as.double} and set as the return value of \texttt{getLatexCharMetrics}.

11 On the Importance of Font and Style Consistency in Reports

If you haven’t figured it out by now, we are quite picky about the way our graphics and reports look. We are especially picky about the consistency in fonts (both sizes and shapes). Without launching into a diatribe about this, we just want to say with tools like \texttt{tikzDevice} you no longer have to settle for what is “just okay.” So go nuts, be picky about how your text and graphics look. Don’t be afraid to snub your nose at reports which pay no attention to detail. Be that person who says “NO! I won’t settle for half rate graphics, I want the best!”

12 The pgfSweave Package and Automatic Report Generation

Now for a little shameless self promotion. The authors of \texttt{tikzDevice} have another package called \texttt{pgfSweave} which provides a driver for Sweave. \texttt{pgfSweave} started as an interface to \texttt{eps2pgf} and its ability to interpret strings in eps files as \texttt{\LaTeX}. This was used to much the same effect as \texttt{tikzDevice}. The problem was the conversion from eps to pgf was SLOW. Long story short, by combining this functionality with the externalization feature of \texttt{pgf} and the \texttt{cacheSweave} we were able to achieve bearable compilation speed and nice looking graphics. \texttt{pgfSweave} is in the process of getting pumped up by interfacing with the \texttt{tikzDevice} package. We hope that the combination will be a self-caching, consistency-inducing, user-empowering tool for high quality reports.
References


R Development Core Team (2009), R Internals: Version 2.9.1.