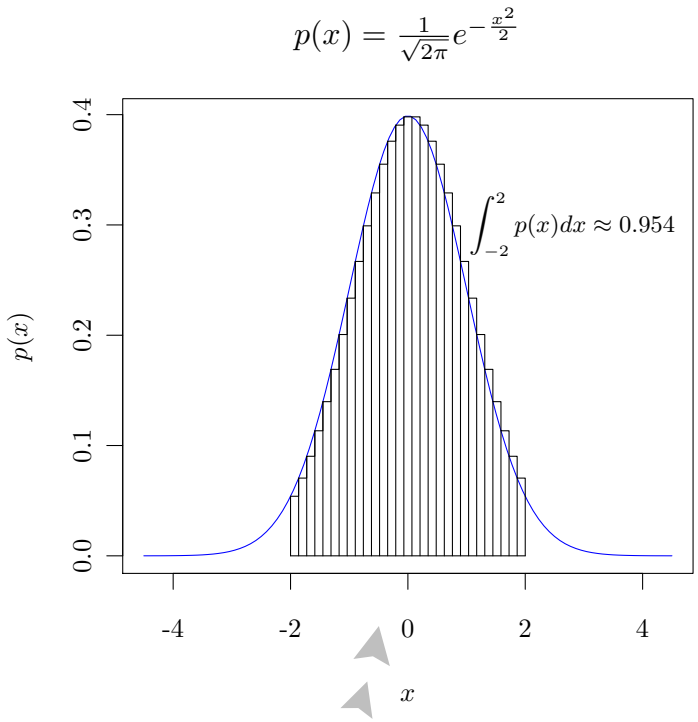


TikZ Device

LaTeX Graphics for R

The **tikzDevice** Package

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

Charlie Sharpsteen and Cameron Bracken

Version 0.5.0-**Beta** February, 2010

Contents

Caveat Utilitor

This is a friendly reminder that the **tikzDevice** package is currently considered by it's designers to be a **beta work**. This package has been released in the hopes that it will help users produce exceptional graphics, leading to an increase in the quality of reports and other materials using those graphics. It has been used for over 6 months by the authors in order to produce figures for academic reports and has proven to be reasonably stable.

The package will remain in Beta for some period of time to allow the authors to consider design decisions underlying the package. **This means the authors are currently reserving the right to alter the package interface in ways that may break existing code that uses the package.** The stability of the package interface is a priority to us but currently we are allowing other concerns to outrank it. When we commit to maintaining a given package structure, the beta flag will be removed and consistency of interface will become a top priority.

During the beta period the actual implementation of the package may differ significantly from what is described in this documentation.

Thanks for giving it a try!
–The tikzDevice Team

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 Ti*k*Z, 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 output.

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 Ti*k*Z device device requires the user to have a working LaTeX compiler along with an installed version of the Ti*k*Z 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 Ti*k*Z 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 TeX 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 driver, created by Valerio Aimale. Without access to such a concise, compact example of implementing a R driver we likely would have abandoned the project in it's 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 it's 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/Ti*k*Z 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 Ti*k*Z 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:

```
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
```

```
Global option tikzLatex set to:
        /usr/texbin/latex
```

If a working LaTeX compiler cannot be found, the **tikzDevice** package will fail to load and a warning message will be displayed:

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

The following places were tested for a valid LaTeX compiler:

        A pre-existing value of the global option tikzLatex
        The R environment variable R_LATEXCMD
        The R environment variable R_PDFLATEXCMD
        The global option latexcmd
        The PATH using the command latex
        The PATH using the command pdflatex
...

Error : .onLoad failed in 'loadNamespace' for 'tikzDevice'
Error: package/namespace load failed for 'tikzDevice'
```

In this case, **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 **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 **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 `.Rprofile` file. All of the options can be set by using `options(<option> = <value>)`. These options allow for the use of custom `documentclass` declarations, LaTeX packages, and typesetting engines (e.g. XeLaTeX). The defaults , if are any for a given option, are shown below the description. The global options are:

`tikzLatex` Specifies the location of the LaTeX compiler to be used by **tikzDevice**. Setting a default for this this option may help the package locate a missing compiler:

Setting the default LaTeX compiler in `.Rprofile`

```
options( tikzLatex = '/path/to/latex/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:

Setting a location in `.Rprofile` for a permanent metrics dictionary

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

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

Default

```
options( tikzDocumentDeclaration = "\\documentclass{article}" )
```

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

Default

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

tikzLatexPackages A character vector. These are the packages which are included when using the `standAlone` option as well as when font metric are calculated.

Default

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

tikzMetricPackages A character vector. 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.

Default

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

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

Default

```
options( tikzSanitizeCharacters = c('%','$','}','{','^') )
```

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

Default

```
options( tikzReplacementCharacters = c('\\%','\\$','\\','\\','\\^'))
```

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 Ti*k*Z 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 just be aware.

4 The `tikz()` Function

4.1 Description

The `tikz()` function provides most of the functionality of the **tikzDevice** package. This function is responsible for creating new R graphics devices that translate the output of graphics functions to the Ti*k*Z 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 `\include{}` function.

4.2 Usage

The `tikz()` function opens a new graphics device and may be called with the following arguments:

```
tikz(file = "Rplots.tex", width = 7, height = 7,
    bg="white", fg="black", standAlone = FALSE, bareBones = FALSE,
    documentDeclaration = getOption("tikzDocumentDeclaration"),
    packages = getOption("tikzLatexPackages"),
    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 `.tex`.

width The width of the output figure, in **inches**.

height The height of the output figure, in **inches**.

bg The starting background color for the plot.

fg The starting foreground color for the plot.

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.

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 `tikzpicture` environment in a LaTeX file.

The next section provides examples of how to use each type of output.

4.3 Examples

4.3.1 Default Mode

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

```
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 `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 `tikz()`. Inside the LaTeX document, the contents of the file `simpleEx.tex` are imported using the `\include{}` command.

Example LaTeX Document

```
\documentclass{article}

% All LaTeX documents including
% tikz() output must use this
% package!
\usepackage{tikz}

\begin{document}
  \begin{figure}[!h]
    \centering

    % The output from tikz()
    % is imported here.
    \input{simpleEx.tex}

    \caption{Simple Example}
  \end{figure}
\end{document}
```

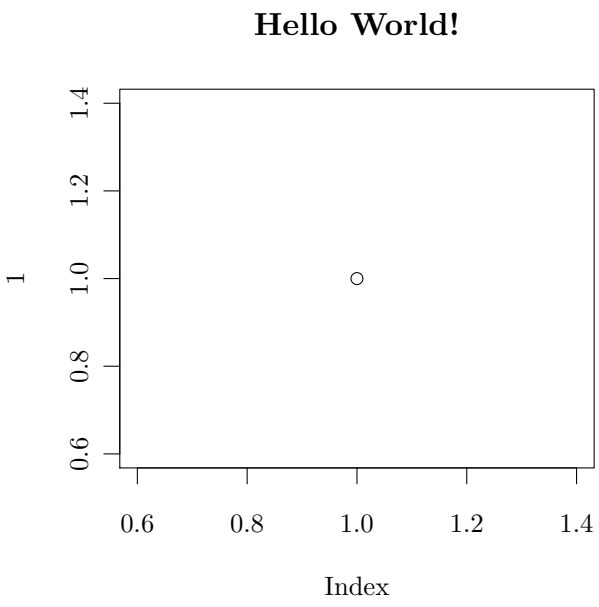


Figure 1: Example of simple `tikz()` usage.

One of the most exciting aspects of the `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 pare prefixed by the backaslash, \, character. This character has a special meaning as an escape charcter in many computing applications, including R. Therefore, it is necessary to place two backslashes, \\, in the input to 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()
```

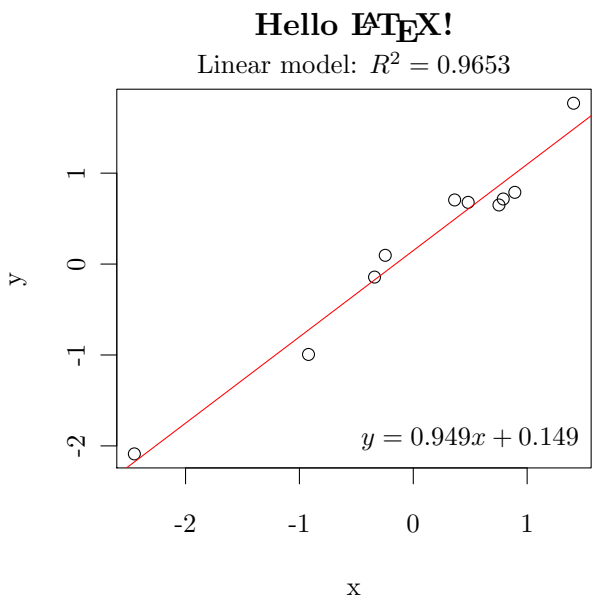


Figure 2: A more complicated example of `tikz()` usage incorporating natively rendered LaTeX commands.

4.3.2 `bareBones` Mode

The `barBones` is designed to facilitate inclusion of `tikz()` output as part of a larger 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 a `tikzpicture` of the users own construction.

```
require(tikzDevice)
tikz('bareBonesExample.tex',width=4,height=4,bareBones=T)
x <- c(0,cumsum(rnorm(25)))
y <- c(0,cumsum(rnorm(25)))
plot(x, y, type="l" )
title("A Brownian Motion")
dev.off()
```

The resulting code may then be used inside a `tikzpicture` environment using the `\include{}` command. The included code must be wrapped in a `scope` environment that contains the options `x=1pt` and `y=1pt`. This informs Ti*k*Z 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`:

Example of a Ti*k*Z environment including `bareBones` output

```
\begin{tikzpicture}

  \draw[clip] (-0.25,-0.25) rectangle (4.25in,4.25in);

  \begin{scope}[x=1pt,y=1pt,yshift=0.25in]
    \input{figs/bareBonesExample}
  \end{scope}

  \node[anchor=south west,draw,rounded corners] (start) at (0,0)
    {TikZ can draw Brownian Motions as Well!};

  \coordinate (current point) at (start.east);
  \coordinate (old velocity) at (0,-5);
  \coordinate (new velocity) at (rand,rand);

  \foreach \i in {25,26,...,50} {

    \draw[blue!\i,ultra thick,line cap=round,x=1cm,y=1cm]
      (current point) .. controls ++([scale=-1]old velocity)
        and ++(new velocity) ..
      ++(rand, rand) coordinate(current point);

    \coordinate(old velocity) at (new velocity);
    \coordinate(new velocity) at (rand,rand);

  }

\end{tikzpicture}
```

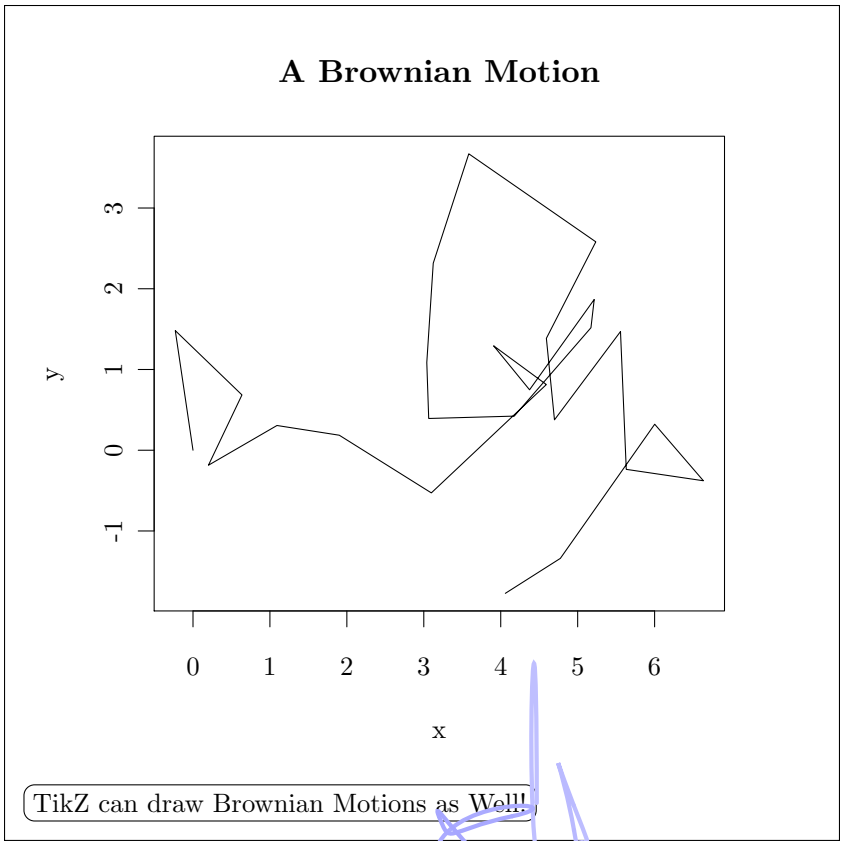


Figure 3: A Ti*k*Z drawing with embedded output from `tikz(bareBones=T)`.

4.3.3 `standAlone` Mode

When the `standAlone` option is passed to `tikz()`, the resulting `.tex` file will be a complete LaTeXdocument designed to be compiled on it's 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:

```
require(tikzDevice)
tikz('standAloneExample.tex',standAlone=T)
plot(sin,-pi,2*pi,main="A Stand Alone TikZ Plot")
dev.off()
```

Note that files produced using the `standAlone` option should not be included in LaTeXdocuments using the `\input{}` command! Use `\includegraphics{}` or load the `pdfpages` package and use `\includepdf{}`.

4.3.4 `console` output Mode

Version 0.5.0 of **tikzDevice** introduced the `console` option. Instead of sending its output to a file `tikz` will send it output to `stdout`. This kind of output can be redirected to a file with `sink()` or spit out directly

into a TEX document from a Sweave file so that the TEX file is self contained.

consoleExample.Rnw

```
\documentclass{article}
\usepackage{tikz}
\usepackage[nogin]{Sweave}
\begin{document}
\begin{figure}[ht]
\centering
<<inline,echo=F,results=tex>>=

require(tikzDevice)
tikz(console=T,width=5,height=5)
  x <- rnorm(100)
  plot(x)
dummy <- dev.off()

@
\caption{caption}
\label{fig:inline}
\end{figure}
\end{document}
```

4.3.5 Getting around special LATEXcharacters

4.3.6 Using the Global Options: An XeLATEX example

It is also possible to use other typesetting engines like XeLATEX by using the global options provided by **tikzDevice**. The following example was inspired by Dario Taraborelli and his article The Beauty of LaTeX.

XeLATEX Example

```
#Set options for using XeLaTeX
options(tikzLatex = 'xelatex')
options(tikzDocumentDeclaration = '\\documentclass{article}')
# The preview package must be loaded first with the xetex driver option
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}"
,"\\usepackage{color}"
,"\\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{\\redc}[1]{\\color{myred}\\fontspec[Variant=4]{Zapfino}#1}"
,"\\newcommand{\\redd}[1]{\\color{myred}\\fontspec[Variant=5]{Zapfino}#1}"
,"\\newcommand{\\rede}[1]{\\color{myred}\\fontspec[Variant=6]{Zapfino}#1}"
,"\\newcommand{\\redf}[1]{\\color{myred}\\fontspec[Variant=7]{Zapfino}#1}"
,"\\newcommand{\\redg}[1]{\\color{myred}\\fontspec[Variant=8]{Zapfino}#1}"
,"\\newcommand{\\lbl}[1]{\\color{lightblue} #1}"
,"\\newcommand{\\lbla}[1]{\\color{lightblue}\\fontspec[Variant=2]{Zapfino}#1}"
,"\\newcommand{\\lblb}[1]{\\color{lightblue}\\fontspec[Variant=3]{Zapfino}#1}"
,"\\newcommand{\\lblc}[1]{\\color{lightblue}\\fontspec[Variant=4]{Zapfino}#1}"
,"\\newcommand{\\lbld}[1]{\\color{lightblue}\\fontspec[Variant=5]{Zapfino}#1}"
,"\\newcommand{\\lble}[1]{\\color{lightblue}\\fontspec[Variant=6]{Zapfino}#1}"
,"\\newcommand{\\lblf}[1]{\\color{lightblue}\\fontspec[Variant=7]{Zapfino}#1}"
,"\\newcommand{\\lblg}[1]{\\color{lightblue}\\fontspec[Variant=8]{Zapfino}#1}"
,"\\newcommand{\\old}[1]{"
,"\\fontspec[Ligatures={Common, Rare},Variant=1,%
Swashes={LineInitial, LineFinal}]{Zapfino}"
,"\\fontsize{25pt}{30pt}\\selectfont #1}%"
,"\\newcommand{\\smallprint}[1]{\\fontspec{Hoefler Text}\\fontsize{10pt}{13pt}"%
,"\\color{Gray}\\selectfont #1}%\n"
))

#Set the content using custom defined commands
label <- c(
  "\\noindent{\\red d}roo{\\lbl g}"
  ,"\\noindent{\\reda d}roo{\\lbla g}"
  ,"\\noindent{\\redb d}roo{\\lblb g}"
  ,"\\noindent{\\redf d}roo{\\lblf g}\\\\[.3cm]"
  ,"\\noindent{\\redc d}roo{\\lblc g}"
  ,"\\noindent{\\redd d}roo{\\lbld g}"
  ,"\\noindent{\\rede d}roo{\\lble g}"
  ,"\\noindent{\\redg d}roo{\\lblg g}\\\\[.2cm]")

#Set the titles using custom defined commands, and hyperlinks
title <- c(
    "\\smallprint{D. Taraborelli (2008),
      \\href{http://nitens.org/taraborelli/latex}%","{The Beauty of \\LaTeX}}"
    ,"\\smallprint{\\\\\\emph{Some rights reserved}.%
    ,"\\href{http://creativecommons.org/licenses/by-sa/3.0/}
      {\\textsc{cc-by-sa}}}"
)

#Draw the graphic
lim <- 0:(length(label)+1)
tikz('xelatexEx.tex',standAlone=T,width=5,height=5)
plot(lim,lim,cex=0,pch='.',xlab = 'Xe\\LaTeX{} Test',
    ylab='', main = title[1], sub = title[2])
```

```
for(i in 1:length(label))
    text(i,i,label[i])
dev.off()
```

Compiling the resulting file `xelatexEx.tex` like so:

Compileing with XeLaTeX

```
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.

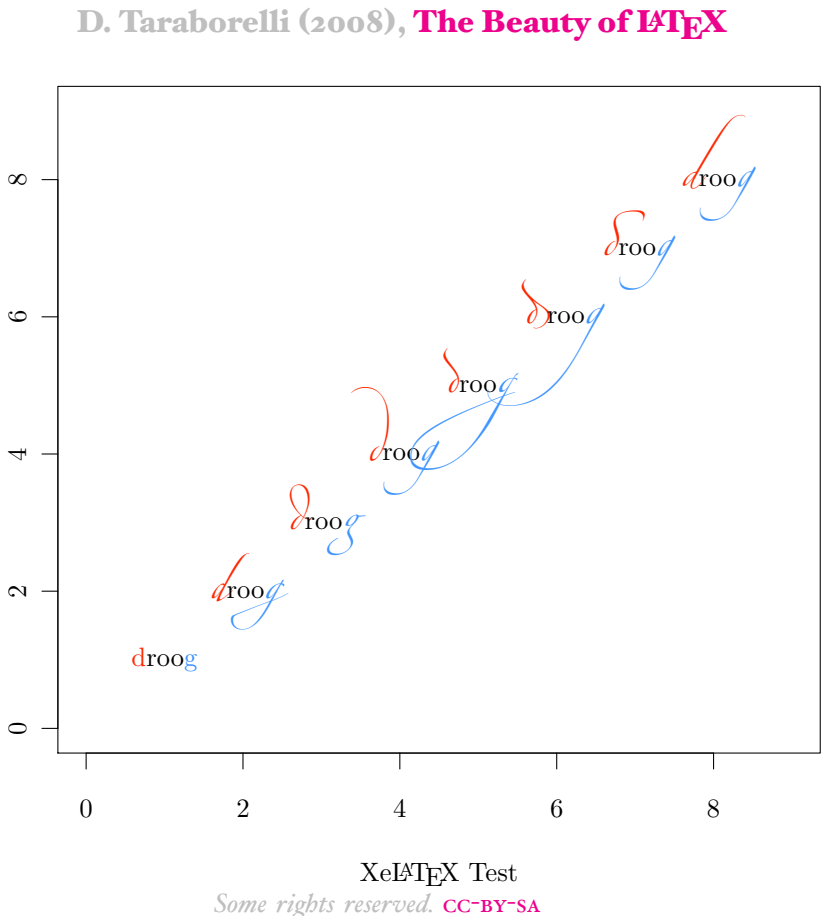


Figure 4: Result of XeLaTeX example

4.3.7 Annotating Graphics with Ti*k*Z Commands

The function `tikzAnnotate` provides the ability to annotate you graphics with Ti*k*Z 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 Ti*k*Z. Check out the results in Figure 5.

```
options(tikzLatexPackages =
    c(getOption('tikzLatexPackages'),
        c("\\usetikzlibrary{decorations.pathreplacing}",
        "\\usetikzlibrary{shapes.arrows}")))
p <- rgamma(300,1)
outliers <- which( p > quantile(p,.75)+1.5*IQR(p) )
tikz("annotation.tex",width=4,height=4)
boxplot(p)
min.outlier <- grconvertY(min( p[outliers] ),, "device")
max.outlier <- grconvertY(max( p[outliers] ),, "device")
x <- grconvertX(1,,"device")
tikzAnnotate(paste("\\coordinate (min) at (",x,',',min.outlier,") {};"))
tikzAnnotate(paste("\\coordinate (max) at (",x,',',max.outlier,") {};"))
tikzAnnotate(c("\\draw[decorate,very thick,red,",
    "decoration={brace,amplitude=20pt}] (min) ",
    "-- node[single arrow,anchor=tip,fill=white,left=20pt,draw=green] ",
    "{Look at These Outliers!} (max);"))
tikzAnnotate(c("\\node[starburst, fill=green, ",
    "draw=blue, very thick,right=of max]  (burst) {Wow!};"))
tikzAnnotate(c("\\draw[->, very thick] (burst.west) -- (max);"))
dev.off()
setTikzDefaults()
```

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 **tikzDevice** package. Cached values of the metrics are returned if they have been calculated by the **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:

```
getLatexStrWidth( texString, cex = 1, face= 1)

getLatexCharMetrics( charCode, cex = 1, face = 1 )
```

`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.

`charCode` An integer between 32 and 126 which indicates a printable character in the ASCII symbol table using the T1 font encoding.

Figure 5: An example using Ti*k*Z annotation.

cex The character expansion factor to be used when determining metrics.

face An integer specifying the R font face to use during metric calculations. The accepted values are as follows:

1: Text should be set in normal font face.

2: Text should be set in **bold font face**.

3: Text should be set in *italic font face*.

4: Text should be set in ***bold italic font face***.

5: Text should be interpreted as `plotmath` symbol characters. Requests for font face 5 are currently ignored.

5.3 Examples

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

```
getLatexStrWidth("The symbol: alpha")
[1] 82.5354

getLatexStrWidth("The symbol: $\\alpha$")
[1] 65.08636
```

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 `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.

```
# 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()`.

```
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 `tlmgr` package manager was introduced in the 2008 distribution. Using `tlmgr` greatly simplifies the adding and removing packages from the distribution. The website offers an installation package, called `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 `tlmgr` utility to add desired add-on packages.

Adam R. Maxwell has created a very nice graphical interface to `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 Ti*k*Z and Other Packages

Unsurprisingly, **tikzDevice** requires the Ti*k*Z package to be installed and available in order to function properly. Ti*k*Z is an abstraction of a lower-level graphics language called PGF and both are distributed as the the `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, `mpm`, can also be run manually from the command prompt:

Using `mpm` to install packages

```
mpm --install packagename
```

For versions of TeX Live and Mac TeX dated 2008 or newer, the `tlmgr` package manager is used in an almost identical manner:

Using `tlmgr` to install packages

```
tlmgr install packagename
```

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 LaTeXpackages is the Comprehensive TeX Archive Network, or CTAN located at http://www.ctan.org. In the case of the PGF/Ti*k*Z package, the project homepage at 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 `texmf` folder. Most TeX distributions allow for each user to have their own personal `texmf` folder somewhere in their home path. The most usual locations, and here ***usual*** is and unfortunately loose term, are as follows:

For UNIX/Linux

```
~/texmf
```

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.
mo
```

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. Fortunately, most packages available on CTAN are archived in such a way that they will unpack into a TDS-complient configuration. TDS-complient archives usually have the phrase `tds` somewhere in their filename and may be installed from a UNIX shell[1] 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-complient 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
```

[1] Sorry 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

Caveat Lector

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.

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** its self 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, Ti*k*Z , 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 a a clipping boundary for any device output that follows.

`line` 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:

Ascent

Ascent is the distance between the baseline and the tallest point on a character's glyph. For the "A" printed to the left, the ascent has been calculated as: 27.33325pt

Descent

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

Width

Width is the distance between the left and right sides of a character's glyph. For the "X" printed to the left, the width has been calculated as: 30.0pt

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 **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: α.

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:

Function declaration for `metricInfo`

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

The most important variables involved in the function are `c`, `ascent`, `descent` and `width`. The incoming variable is `c`, which contains the character for which R is requesting font metrics. Interestingly, `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 `ascent`, `descent` and `width`. The asterisks, '*', 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 `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 `return` statement. So, the main task of the `metricInfo` function is to assign values to `ascent`, `descent` and `width`.

The other two variables present in the function are the `pGEcontext` variable `gc` and the `pDevDesc` variable `dd`. `gc` contains information such as the font face, foreground color, background color, character expansion factor, ect. currently in use by the graphics system. `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 `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 `ascent`, `descent` and `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, `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 *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 `SEXP`. For the exact details on what a `SEXP` is and how it works, we refer the interested user to chapter one of the *R Internals* manual.

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

Definition of target R function

```
getLatexCharMetrics <- function( charCode ){

  # System call to LaTeX

}
```

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 `charCode`. This is set up in C as follows:

Preparing a R function call inside C

```
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 ) );
  SET_TAG( CDR( RCallBack ), install("charCode") );

  \\ To be continued...

}
```

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:

```
SETCAR( 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.

Executing a R function call inside C

```
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`?

Recovering return values from a R function call

```
  *ascent = REAL(RMetrics)[0];
  *descent = REAL(RMetrics)[1];
  *width = REAL(RMetrics)[2];

  UNPROTECT(2);

  return;
```

Here the `REAL` function is used to coerce the `SEXP` variable `RMetrics` to a vector of real numbers. These numbers are then extracted and assigned to the return values of `metricInfo`. In C we must specify the 'first' value in a vector using the index 0 rather than the index 1 [2]. The last thing to do is release the restrictions we placed on the R garbage collector. Since we used the `PROTECT` function twice, we must call `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 R function `getLatexCharMetrics`. The first thing we need to do is set up a file for LaTeX input:

Creating a LaTeX input file

```
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 `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 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. 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 LaTeXfile.

[2] 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.

Setting up the preamble of a LaTeX input file

```
writeLines("\\documentclass{article}", texIn)

writeLines("\\usepackage[T1]{fontenc}", texIn)

writeLines("\\usepackage{tikz}", texIn)
writeLines("\\usetikzlibrary{calc}", texIn)

writeLines("\\batchmode", texIn)
```

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 TeXand 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, R expects us to be using the Type 1 encoding (R 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 Ti*k*Z 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 Ti*k*Z 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`.

Extracting character dimensions using Ti*k*Z

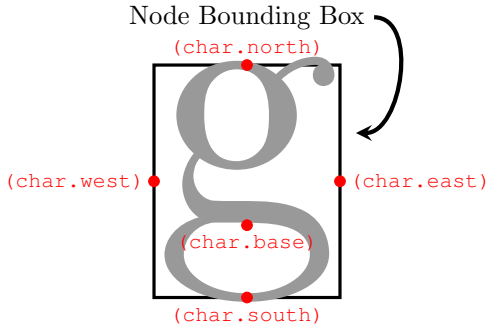
```
\begin{tikzpicture}

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

\path let \p1 = ($(char.east) - (char.west)$),
  \n1 = {veclen(\x1,\y1)} in (char.east) -- (char.west)
  node{ \typeout{tikzTeXWidth=\n1} };

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

\path let \p1 = ($(char.base) - (char.south)$),
  \n1 = {veclen(\x1,\y1)} in (char.base) -- (char.south)
  node{ \typeout{tikzTeXDescent=\n1} };
```

What the heck just happened? Well, first we instructed LaTeX to enter the Ti*k*Z picture environment using `\begin{tikzpicture}`. Then we ordered Ti*k*Z to create a node named "char" containing the command

\char 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 inner sep and outer sep options are set to 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*k*Z defines 'anchors' along the bounding box of the node. All anchors are referred using a node name.posistion notation. Since we named the node char, all the anchors start with char. The anchor posistions relevant to our problem are shown below:



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 char.north and char.base. Similarly, for the descent we would calculate the difference in height between char.base and char.south and width can be obtained using char.west and char.east. This is the purpose of the admittedly cryptic \path commands that are inserted in the LaTeX input file. Let's examine one of them:

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

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

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

Draws a line from (0,0) to (1,1) and places a node at (1,1) containing the word 'Hi!'. In the Ti*k*Z code produced by getLatexCharMetrics, the let operation is specified. Basically, let postpones the actual drawing of a path and performs calculations until the in keyword is encountered. The result of these calculations are stored in a set of special variables which must start with \n, \p, \x or \y. The first let operation executed is:

```
\p1 = ( $(char.north) - (char.base)$ )
```

This performs a vector subtraction between the coordinates of `char.north` and `char.base`. The resulting x and y components are stored in the 'point' variable `\p1`. The second operation executed is:

```
\n1 = {veclen(\x1,\y1)}
```

This code `let` operation treats the coordinates stored in `\p1` as a vector and calculates its magnitude. The '1' appended to the `\x` and `\y` variables specifies that we are accessing the x and y components of `\p1`. This result is stored in the 'number' variable `\n1`. Now, that our metric is stored in `\n1`, our final task is to ensure it makes it into the LaTeX `.log` file– this is done by adding a `node` containing the `\typeout` command. The contents of the node:

```
\typeout{tikzTexAscent=\n1}
```

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:

Terminating a LaTeX compilation

```
writeLines("\\makeatother", texIn)

writeLines("\\@@end", texIn)

close(texIn)
```

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 `\typeout`.

Terminating a LaTeX compilation

```
latexCmd <- getOption('tikzLatex')
latexCmd <- 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 )
```

The LaTeX compiler is executed through the `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 `system`

function to a dummy variable called `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.

Parsing the `.log` file text

```
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 `grep` function to search through the log output for the tags 'tikzTeXWidth=', 'tikzTeXAscent=' and 'tikzTeXDescent=' that we specified when we used `\typeout`. After we recover a line containing one of these tags, we use the `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 `as.double` and set as the return value of `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 **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 wont 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 **tikzDevice** have another package called **pgfSweave** which provides a driver for Sweave. **pgfSweave** started as an interface to eps2pgf and its ability to interpret strings in eps files as LaTeX. This was used to much the same effect as **tikzDevice**. The problem was the conversion from eps to pgf was SLOW. Long story short, by combining this functionality with the externalization feature of pgf and the **cacheSweave** we were able to achieve bearable compilation speed and nice looking graphics. **pgfSweave** is in the process of getting pumped up by interfacing with the **tikzDevice** package. We hope that the combination will be a self-caching, consistency-inducing, user-empowering tool for high quality reports.

References

Murrell, P. (2005), Using Computer Modern Fonts in R Graphics, http://www.stat.auckland.ac.nz/~paul/R/CM/CMR.html.

Peng, R. D. (2006), Interacting with data using the filehash package, *R News*, *6*(4), 19–24.

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

Tantau, T. (2008), *The TikZ and PGF Packages: Manual for version 2.00*.