

# TikZ Device

## L<sup>A</sup>T<sub>E</sub>X Graphics for R

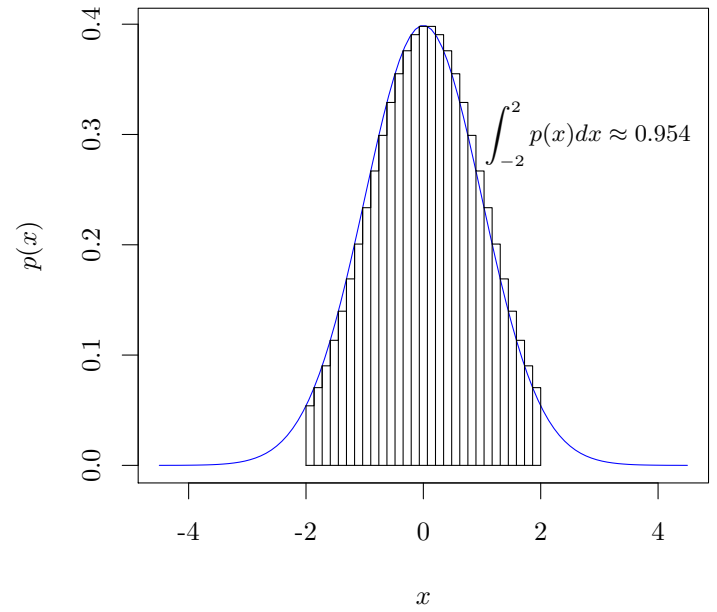
Charlie Sharpsteen  
Cameron Bracken

Version: 0.12.3.1

NA

<https://github.com/yihui/tikzDevice>

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



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Acknowledgements . . . . .	2
<b>I</b>	<b>Usage and Examples</b>	<b>3</b>
<b>2</b>	<b>Loading the Package</b>	<b>4</b>
2.1	Options That Affect Package Behavior . . . . .	5
	The tikzDefaultEngine Option . . . . .	5
	The tikzLatex, tikzXelatex and tikzLualatex Options . . . . .	5
	The tikzMetricsDictionary Option . . . . .	5
	The tikzDocumentDeclaration Option . . . . .	6
	The tikzLatexPackages, tikzXelatexPackages and tikzLualatexPackages Options . . . . .	6
	The tikzMetricPackages and tikzUnicodeMetricPackages Options . . . . .	7
	The tikzFooter Option . . . . .	7
	The tikzSanitizeCharacters and tikzReplacementCharacters Options . . . . .	7
	The tikzLwdUnit Option . . . . .	7
	The deprecated tikzRasterResolution Option . . . . .	8
	The tikzPdftexWarnUTF Option . . . . .	8
<b>3</b>	<b>The tikz Function</b>	<b>8</b>
3.1	Description . . . . .	8
3.2	Usage . . . . .	8
3.3	Font Size Calculations . . . . .	9
	UTF-8 Output . . . . .	10
3.4	Examples . . . . .	10
	Default Mode . . . . .	10
	bareBones Mode . . . . .	12
	standAlone Mode . . . . .	14
	console output Mode . . . . .	15
	Using Xe <sup>ℒ</sup> TeX . . . . .	15
	Annotating Graphics with TikZ Commands . . . . .	17
	tikz vs. pdf for plotmath symbols and Unicode characters . . . . .	20
<b>4</b>	<b>The getLatexCharMetrics and getLatexStrWidth Functions</b>	<b>21</b>
4.1	Description . . . . .	21
4.2	Usage . . . . .	21
4.3	Examples . . . . .	21
<b>II</b>	<b>Installation Guide</b>	<b>24</b>
<b>5</b>	<b>Obtaining a <sup>ℒ</sup>TeX Distribution</b>	<b>25</b>
5.1	Windows . . . . .	25
5.2	UNIX/Linux . . . . .	25
5.3	Mac OS X . . . . .	25
5.4	Installing TikZ and Other Packages . . . . .	25
	Using a <sup>ℒ</sup> TeX Package Manager . . . . .	26
	Manual Installation . . . . .	26

---

<b>III Package Internals</b>	<b>28</b>
<b>6 Introduction and Background</b>	<b>29</b>
<b>7 Anatomy of an R Graphics Device</b>	<b>29</b>
7.1 Drawing Routines . . . . .	29
7.2 Font Metric Routines . . . . .	30
7.3 Utility Routines . . . . .	30
<b>8 Calculating Font Metrics</b>	<b>30</b>
Character Metrics . . . . .	31
Calling R Functions from C Functions . . . . .	31
Implementing a System Call to $\text{\LaTeX}$ . . . . .	33
<b>9 On the Importance of Font and Style Consistency in Reports</b>	<b>37</b>
9.1 The <code>pgfSweave</code> Package and Automatic Report Generation . . . . .	37
<b>Bibliography</b>	<b>38</b>

# Introduction

The **tikzDevice** package provides a graphics output device for R that records plots in a L<sup>A</sup>T<sub>E</sub>X-friendly format. The device transforms plotting commands issued by R functions into L<sup>A</sup>T<sub>E</sub>X code blocks. When included in a paper typeset by L<sup>A</sup>T<sub>E</sub>X, these blocks are interpreted with the help of TikZ—a graphics package for T<sub>E</sub>X and friends written by Till Tantau. By allowing L<sup>A</sup>T<sub>E</sub>X to handle typesetting of text in R plots along with the rest of the text in the paper the full power of T<sub>E</sub>X is available to the R user. There are also no discontinuities in font selection and typesetting as demonstrated by a comparison between [Figure 1](#) and [Figure 2](#).

```
pdf('pdf-example.pdf',
    width = 3.25, height = 3.25)
plot(1, 1, main = 'Hello!')
dev.off()
```

**Hello!**

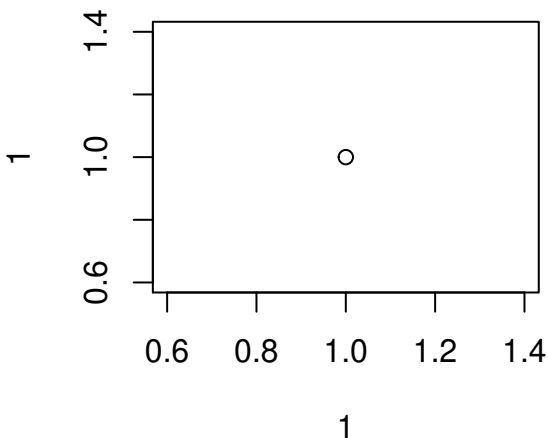


Figure 1: Output from pdf()

```
tikz('tikz-example.tex',
     width = 3.25, height = 3.25)
plot(1, 1, main = 'Hello \TeX !')
dev.off()
```

**Hello T<sub>E</sub>X!**

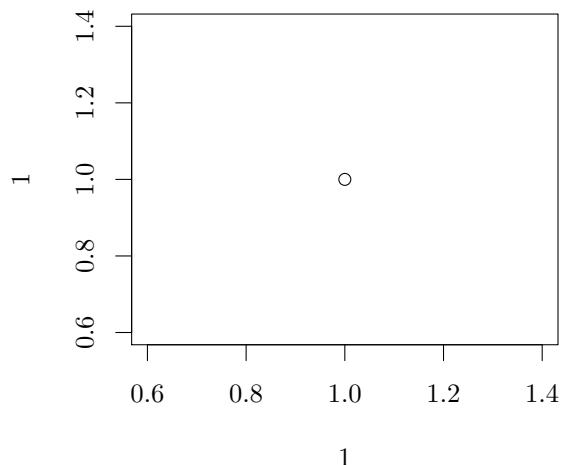


Figure 2: Output from tikz()

This document is divided into three parts. [Part I](#) describes the package functionality and provides example usage. Besides the R environment, use of the TikZ device requires the user to have a working L<sup>A</sup>T<sub>E</sub>X compiler along with an installed version of the TikZ package—version 2.00 or greater. [Part II](#) of this documentation offers suggestions on how to get these dependencies installed and working properly.

[Part III](#) is intended for those who are curious as to the details of how this package is implemented. This part attempts to explain how the **tikzDevice** 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 information is provided in the hope that the **tikzDevice** may serve as a case study for creating new R graphics devices. 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.

## 1.1 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  $\text{\TeX}$  typesetting system. It is our hope that this work will make it easier for users 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  $\text{PicTeX}$  device created by Valerio Aimale which is part of the R core graphics system. Without access to this simple, compact example of implementing a graphics device we likely would have abandoned the project in its infancy. We would also like to thank Paul Murrell 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  $\text{\TeX}$  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  $\text{\LaTeX}$  graphics package that makes the output of this device meaningful. We would also like to express deep appreciation for the beautiful 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!

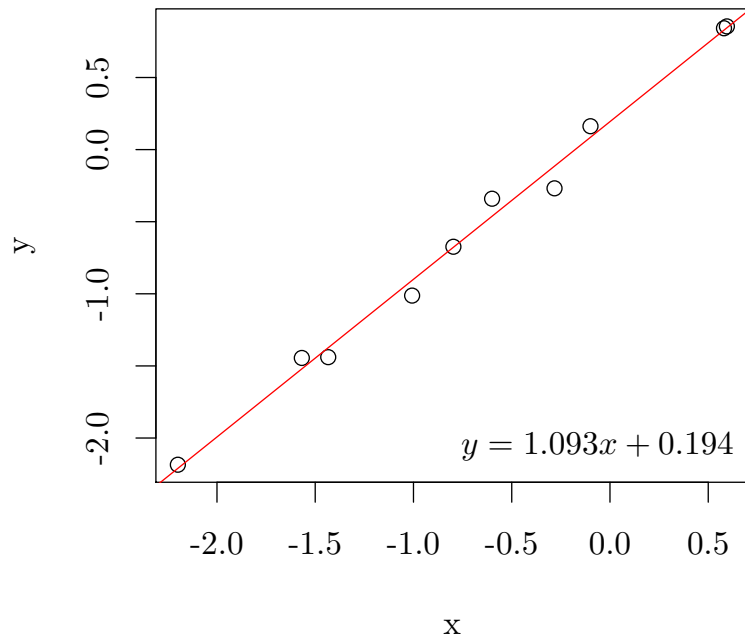
—*The tikzDevice Team*

# Part I

## Usage and Examples

**Hello L<sup>A</sup>T<sub>E</sub>X!**

Linear model:  $R^2 = 0.9927$



## Loading the Package

The functions in the **tikzDevice** package are made accessible in the R environment by using `library()`:

```
library(tikzDevice)
```

Upon loading, the package will search for the following  $\text{\LaTeX}$  compilers:

- $\text{pdf\LaTeX}$
- $\text{Xe\LaTeX}$
- $\text{Lua\LaTeX}$

Access to  $\text{\LaTeX}$  is essential for the device to produce output as the compiler is queried for font metrics when constructing plots that contain text. For more information on why communication between the device and  $\text{\LaTeX}$  is necessary, see [Part III](#). The package will fail to load if  $\text{pdf\LaTeX}$  cannot be located. The presence of the  $\text{Xe\LaTeX}$  and  $\text{Lua\LaTeX}$  compilers is optional. When the package loads successfully, a startup message will be printed that looks similar to the following:

```
Loading required package: filehash
filehash: Simple key-value database (2.2 2011-07-21)
tikzDevice: R Graphics Output in LaTeX Format (v0.7)
  LaTeX found in the PATH using the command: pdflatex
  XeLaTeX found in the PATH using the command: xelatex
  LuaLaTeX found in the PATH using the command: lualatex
```

If a working  $\text{pdf\LaTeX}$  compiler cannot be found, the **tikzDevice** package will fail to load and a warning message will be displayed:

```
Error : .onLoad failed in loadNamespace() for 'tikzDevice', details:
  call: fun(libname, pkgname)
  error:

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

The following places were tested for a valid LaTeX compiler:

  the global option: tikzLatex
  the environment variable: R_LATEXCMD
  the environment variable: R_PDFLATEXCMD
  the global option: latexcmd
  the PATH using the command: pdflatex
  the PATH using the command: latex
  the PATH using the command: /usr/texbin/pdflatex

...

Error: loading failed
```

In this case, **tikzDevice** has done its 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](#).

## 2.1 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.  $X_{\text{q}}\LaTeX$  or  $\text{Lua}\LaTeX$ ).

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 the R manual page `?Startup`. For the details of why calling the  $\LaTeX$  compiler is necessary, see [Part III](#).

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.

### The **tikzDefaultEngine** Option

This option specifies which typesetting engine the **tikzDevice** package will prefer. Current possible values are `pdftex`, `xetex` or `luatex` which will respectively trigger the use of the  $\text{pdf}\LaTeX$ ,  $X_{\text{q}}\LaTeX$  or  $\text{Lua}\LaTeX$  compilers.

#### Default

```
options(tikzDefaultEngine = 'pdftex')
```

#### Choosing the $\TeX$ engine

```
options(tikzDefaultEngine = 'xetex')
options(tikzDefaultEngine = 'luatex')
```

### The **tikzLatex**, **tikzXelatex** and **tikzLualatex** Options

Specifies the location of the  $\LaTeX$ ,  $X_{\text{q}}\LaTeX$  and  $\text{Lua}\LaTeX$  compilers to be used by **tikzDevice**. Setting a default for this option may help the package locate a missing compiler:

#### Setting default compilers in `.Rprofile`

```
options(tikzLatex = '/path/to/pdflatex')
options(tikzXelatex = '/path/to/xelatex')
options(tikzLualatex = '/path/to/luatex')
```

### The **tikzMetricsDictionary** Option

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. Using the option `tikzMetricsDictionary`, a permanent cache file may be specified:



Setting a location in .Rprofile for a permanent metrics dictionary

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

### The **tikzDocumentDeclaration** Option

For `standAlone == TRUE`, `tikzDocumentDeclaration` also influences the calculation of font metrics. If a base font size is specified as an option to the document class, the **tikzDevice** will use this value as the base font size for graphics.

Default

```
options(tikzDocumentDeclaration = "\\documentclass[10pt]{article}")
```

### The **tikzLatexPackages**, **tikzXelatexPackages** and **tikzLualatexPackages** Options

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`, or additional L<sup>A</sup>T<sub>E</sub>X macros, such as `amsmath`, these packages should be added to this list.

Default

```
options(
  tikzLatexPackages = c(
    "\\usepackage{tikz}",
    "\\usepackage[active,tightpage]{preview}",
    "\\PreviewEnvironment{pgfpicture}",
    "\\setlength\\PreviewBorder{0pt}")
),
  tikzXelatexPackages = c(
    "\\usepackage{tikz}\\n",
    "\\usepackage[active,tightpage,xetex]{preview}\\n",
    "\\usepackage{fontspec,xunicode}\\n",
    "\\PreviewEnvironment{pgfpicture}\\n",
    "\\setlength\\PreviewBorder{0pt}\\n"
),
  tikzLualatexPackages = c(
    "\\usepackage{tikz}\\n",
    "\\usepackage[active,tightpage,psfixbb]{preview}\\n",
    "\\usepackage{fontspec,xunicode}\\n",
    "\\PreviewEnvironment{pgfpicture}\\n",
    "\\setlength\\PreviewBorder{0pt}\\n"
)
)
```

Adding a package that affects fonts

```
options( tikzLatexPackages = c(
  getOption( "tikzLatexPackages" ),
  "\\usepackage{mathpazo}"
))
```

## The `tikzMetricPackages` and `tikzUnicodeMetricPackages` Options

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 the  $X_{\text{F}}\text{T}_{\text{E}}\text{X}$  or  $\text{LuaT}_{\text{E}}\text{X}$  engines are in use.

Default

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

  tikzUnicodeMetricPackages = c(
    "\\usepackage[T1]{fontenc}\n",
    "\\usetikzlibrary{calc}\n",
    "\\usepackage{fontspec,xunicode}\n"
  )
)
```

## The `tikzFooter` Option

A character vector. This footer is appended to the end of the figure when `standAlone==TRUE` before the `\end{document}` statement.

Default

```
options(tikzFooter = "")
```

## The `tikzSanitizeCharacters` and `tikzReplacementCharacters` Options

`tikzSanitizeCharacters` is a character vector of special  $\LaTeX$  characters to replace while `tikzReplacementCharacters` is a character vector containing the corresponding replacements.

Default

```
options(
  tikzSanitizeCharacters = c('%', '$', ')', '{', '^', '_', '#', '&', '~'),
  tikzReplacementCharacters = c('\\%', '\\$', '\\}', '\\{', '\\^', '\\_{}', '\\_{}',
  '\\#', '\\&', '\\char`\\~')
)
```

## The `tikzLwdUnit` Option

Originally, 1 unit of line width in R was translated to 0.4 pt. This results in lines that are thinner than with default graphics or with the PDF device. This option controls the translation factor – use `72.27 / 96` for compatibility with the R default (96 pixels in R is 1 inch, which is 72.27 points in  $\text{T}_{\text{E}}\text{X}$ ).

Default

```
options(tikzLwdUnit = 72.27 / 96)
```

## The deprecated **tikzRasterResolution** Option

When `tikz` is requested to add a raster to a graphic, the raster is written to a PNG file which is then included by the  $\LaTeX$  code. In the current version, the raster is always written “as is” (after mirroring has been applied) using `png::writePNG()` (Urbanek, 2013). No resampling or transformation of any kind are applied in this process, rotation and interpolation are carried out by  $\LaTeX$ .

## The **tikzPdfTeXWarnUTF** Option

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

Default

```
options(tikzPdfTeXWarnUTF = TRUE)
```

CHAPTER 3

# The `tikz` Function

## 3.1 Description

The `tikz` function provides most of the functionality of the **tikzDevice** package. This function opens an R graphics device that records plots as a series of TikZ commands. The device supports many levels of output that range from stand-alone  $\LaTeX$  documents that may be compiled into figures to code chunks that must be incorporated into existing  $\LaTeX$  documents using the `\include{}` macro.

## 3.2 Usage

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

```
tikz(file = filename,
      filename = ifelse(onefile, "./Rplots.tex", "./Rplot%03d.tex"), width = 7,
      height = 7, onefile = TRUE, bg = "transparent", fg = "black",
      pointsize = 10, lwdUnit = getOption("tikzLwdUnit"), standAlone = FALSE,
      bareBones = FALSE, console = FALSE, sanitize = FALSE,
      engine = getOption("tikzDefaultEngine"),
      documentDeclaration = getOption("tikzDocumentDeclaration"), packages,
      footer = getOption("tikzFooter"),
      symbolicColors = getOption("tikzSymbolicColors"),
      colorFileName = "%s_colors.tex",
      maxSymbolicColors = getOption("tikzMaxSymbolicColors"), timestamp = TRUE,
      verbose = interactive())
```

`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*.

`onefile` Controls whether output should be directed to a single file containing one `tikzpicture` environment per plot or split into multiple files each containing a single `tikzpicture` environment.

- `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 `getOption("tikzDocumentDeclaration")`. See [Section 3.3](#) for more details.
- `lwdUnit` The number of pts in LaTeX that `lwd=1` in R is translated to. Defaults to 0.4 (LaTeX and TikZ default); for compatibility with R default, please use 72.27/96 (96 pixels in R is 1 inch, which is 72.27 points in TeX). See also [Section 2.1](#), “Options That Affect Package Behavior.”
- `standAlone` A logical value indicating whether the resulting file should be suitable for direct processing by L<sup>A</sup>T<sub>E</sub>X.
- `bareBones` A logical value indicating whether TikZ code is produced without being placed within a `tikzpicture` environment.
- `console` Controls whether output is directed to the R console. This is useful for dumping TikZ output directly into a L<sup>A</sup>T<sub>E</sub>X 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 [Section 2.1](#), “Options That Affect Package Behavior” for which characters are replaced.
- `engine` A string specifying which T<sub>E</sub>X engine to use. Possible values are `'pdftex'`, `'xetex'` and `'luatex'`.
- `documentDeclaration` See [Section 2.1](#), “Options That Affect Package Behavior.”
- `packages` See [Section 2.1](#), “Options That Affect Package Behavior.”
- `footer` See [Section 2.1](#), “Options That Affect Package Behavior.”

The first six options should be familiar to anyone who has used the default graphics devices shipped with R. The 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 L<sup>A</sup>T<sub>E</sub>X files suitable for compilation.
- Graphics production as complete figures suitable for inclusion in L<sup>A</sup>T<sub>E</sub>X files.
- Graphics production as raw figure code suitable for inclusion in an enclosing `tikzpicture` environment in a L<sup>A</sup>T<sub>E</sub>X file.

### 3.3 Font Size Calculations

The overarching goal of the **tikzDevice** is to provide seamless integration between text in R graphics and the text of L<sup>A</sup>T<sub>E</sub>X 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 L<sup>A</sup>T<sub>E</sub>X. The issue is that font sizes in L<sup>A</sup>T<sub>E</sub>X are controlled by a “base font size” that is specified at the beginning of the document—typically 10pt. There is no easy way in L<sup>A</sup>T<sub>E</sub>X 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 `par`.
- The “font size” parameter, specified using the `'ps'` argument to functions such as `par` or the `'fontsize'` argument to functions such as `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 `options( "tikzDocumentDeclaration" )` using the regular expression `\d+pt`. With the default header:

```
\documentclass[10pt]{article}
```

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 `pointsize` argument to the `tikz` function will be used.

## UTF-8 Output

Version 0.6.0 of the **tikzDevice** introduced support for (multibyte) Unicode characters in the text of graphics through support for  $\XeTeX$ . Version 0.7.0 extended multilingual typesetting further by adding support for the  $\LuaTeX$  compiler. Unicode support in **tikzDevice** is subject to a few important caveats:

**System Requirements:** A working version of  $\XeTeX$  or  $\LuaTeX$  along with the packages `fontspec` and `xunicode` are required for direct processing of Unicode input. If **tikzDevice** cannot find a Unicode-aware compiler, then Unicode support cannot be guaranteed.

**Encoding:** **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 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 **tikzDevice** but the examples will not work out of the box.

**Compiling:** A graphic that contains UTF-8 characters should be compiled with  $\XeTeX$  or  $\LuaTeX$  with the `xunicode` and `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).

## 3.4 Examples

### 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:

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

A  $\LaTeX$  document is then required to display the figure. This document must include `TikZ` as one of the packages that it loads. `TikZ` provides several libraries that enable 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.

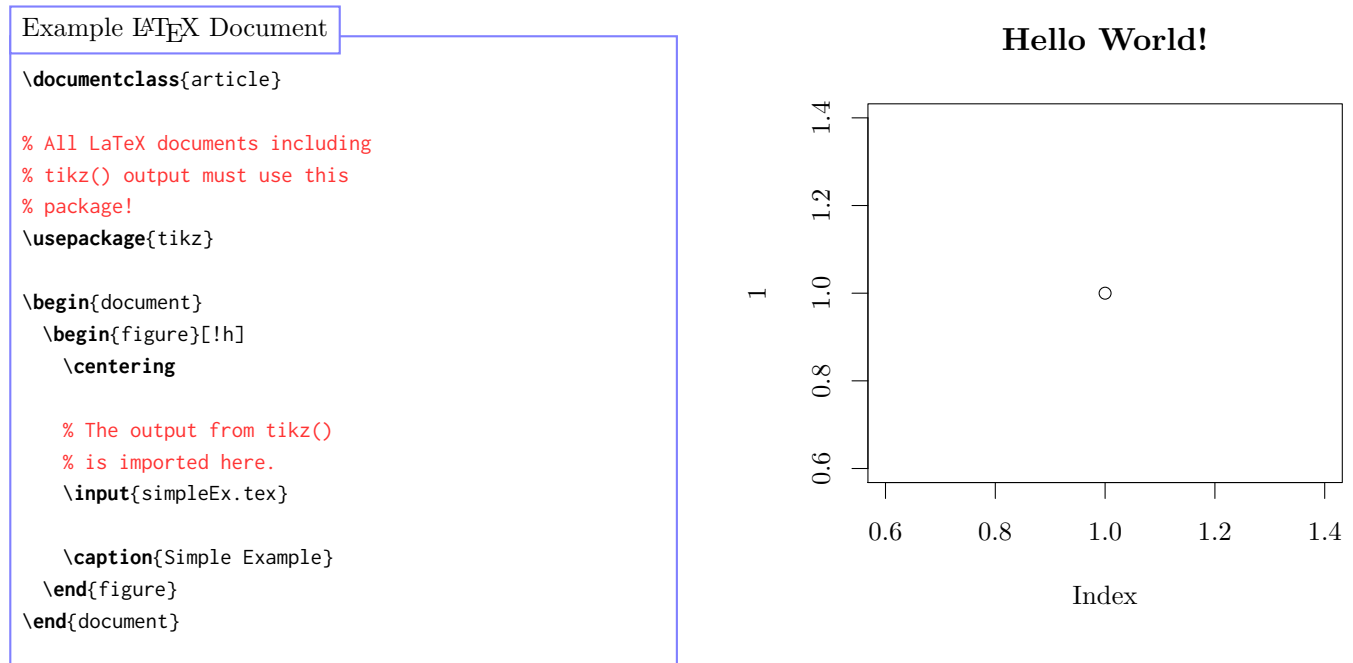


Figure 3: Example of simple `tikz` usage.

One of the most exciting aspects of the `tikz` function is that the inclusion of arbitrary  $\LaTeX$  code is allowed in strings passed to plotting commands. An important issue to note is that many  $\LaTeX$  commands are prefixed by the backslash character: `\`. Unfortunately, in many programming languages, the backslash character is also given a special status when it appears in strings. Therefore, it is necessary to place two backslashes, `\\`, in R strings in order to cause backslash to appear in the output for  $\LaTeX$ . The next example demonstrates how to use  $\LaTeX$  commands in plot annotation.

```

library(tikzDevice)
tikz('figs/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()

```

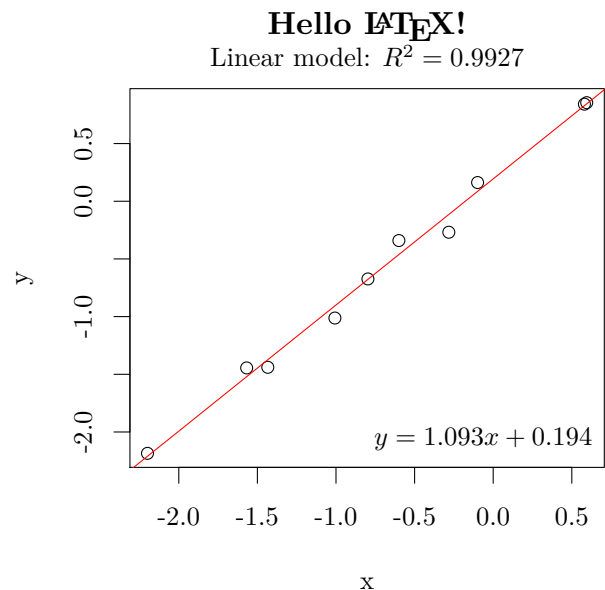


Figure 4: A more complicated example of tikz usage incorporating natively rendered  $\LaTeX$  commands.

## bareBones Mode

bareBones output is designed to facilitate inclusion of code generated by tikz into a larger TikZ graphic. Normally tikz wraps output as a self-contained tikzpicture environment. When bareBones is invoked, the wrapping environment is omitted. This option allows output to be embedded inside another tikzpicture of the user's own construction.

```

library(tikzDevice)
library(maps)

tikz('figs/westCoast.tex', bareBones=TRUE)

map('state', regions=c('california', 'oregon', 'washington'),
     lwd=4, col='grey40')

# Insert some named coordinates into the picture that will
# be available once the picture is included into the
# TeX document.
tikzCoord(-124.161, 40.786, 'humBay')
tikzCoord(-122.962, 46.148, 'longView')
tikzCoord(-124.237, 43.378, 'coosBay')
tikzCoord(-122.419, 37.775, 'sfBay')

dev.off()

```

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`:

Example of a `TikZ` environment including `bareBones` output

```
\begin{tikzpicture}

% Include bareBones output inside a scope with x and y units set to 1pt
\begin{scope}[x=1pt,y=1pt]
  \input{figs/westCoast}
\end{scope}

% Label ports using coordinates placed into the barBones output by the
% tikzAnnotate function.
\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}
```



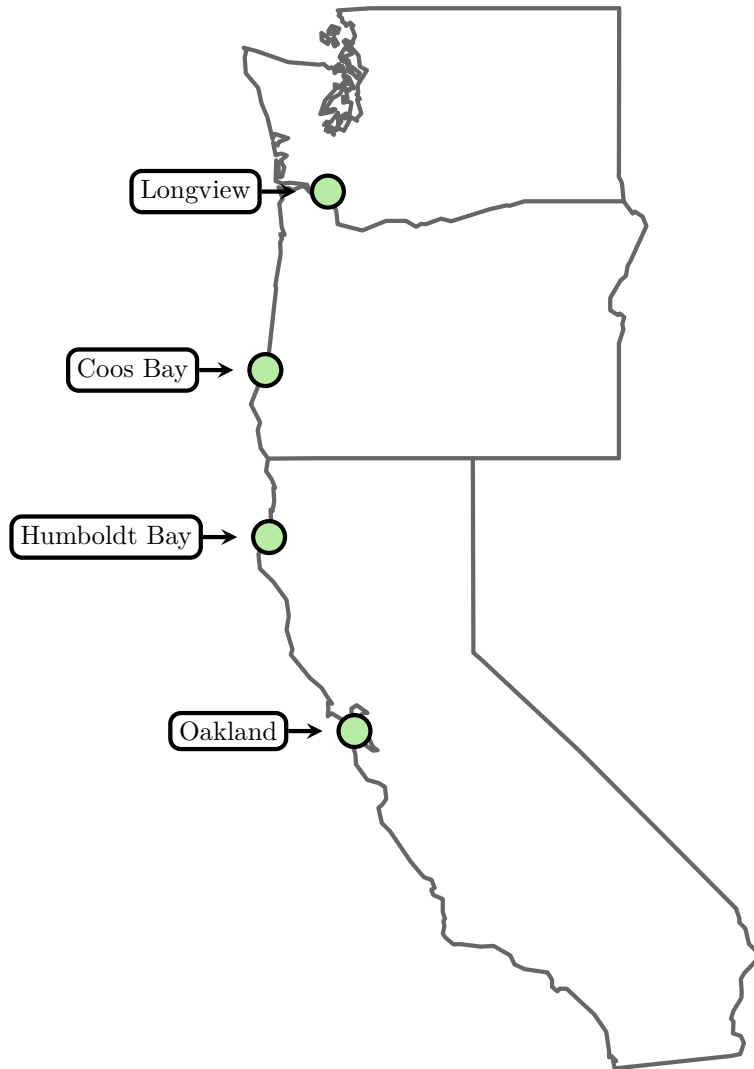


Figure 5: A TikZ drawing with embedded output from `tikz(bareBones=TRUE)`.

### **standAlone** Mode

When the `standAlone` option is passed to `tikz`, the resulting `.tex` file will be a complete  $\LaTeX$  document that can be compiled into a stand-alone figure. 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` to crop the pages in the resulting document to the bounding boxes of the plots. Stand-alone output may be produced in the following manner:

```
library(tikzDevice)
tikz('standAloneExample.tex',standAlone=TRUE)
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  $\LaTeX$  documents using the `\input` command! Use `\includegraphics` or load the `pdfpages` package and use `\includepdf`.

## console output Mode

Version 0.5.0 of `tikzDevice` introduced the `console` option. With this option, `tikz` will send output to `stdout` instead of a file. 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 and does not include other files via `\input`. (Including the chunk option `strip.white=FALSE` was necessary for some versions of `tikzDevice` prior to 0.7.2.)

### Catching tikz output inside Sweave

```
\documentclass{article}
\usepackage{tikz}

\begin{document}
  \begin{figure}[ht]
    \centering
    <<inline,echo=FALSE,results='tex'>>=

    require(tikzDevice)
    tikz(console=TRUE,width=5,height=5)
    x <- rnorm(100)
    plot(x)
    dummy <- dev.off()
  @
    \caption{caption}
    \label{fig:inline}
  \end{figure}
\end{document}
```

## Using Xe $\LaTeX$

It is also possible to use other typesetting engines like Xe $\LaTeX$  by using the global options provided by `tikzDevice`. The following example was inspired by Dario Taraborelli and his article [The Beauty of LaTeX](#).

```
# Set options for using XeLaTeX font variants.
options(tikzXeLatexPackages = c(
  getOption('tikzXeLatexPackages'),
  "\\usepackage[colorlinks, breaklinks]{hyperref}",
  "\\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{\lbl e}[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}"
))

# 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{\lbl d}",
  "\noindent{\rede d}roo{\lbl e}",
  "\noindent{\redg d}roo{\lblg g}\\\\[.2cm]"
)

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

# Draw the graphic
tikz('xelatexEx.tex',
  standAlone=TRUE,width=5,height=5,
  engine = 'xetex')
lim <- 0:(length(label)+1)
plot(lim,lim,cex=0,pch='.',xlab = title[2],ylab='', main = title[1])
for(i in 1:length(label))
  text(i,i,label[i])

```



```

    "\\usetikzlibrary{shapes.arrows,shapes.symbols}")
)

p <- rgamma (300 ,1)
outliers <- which( p > quantile(p,.75)+1.5*IQR(p) )
boxplot(p)

# Add named coordinates that other TikZ commands can hook onto
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()

```

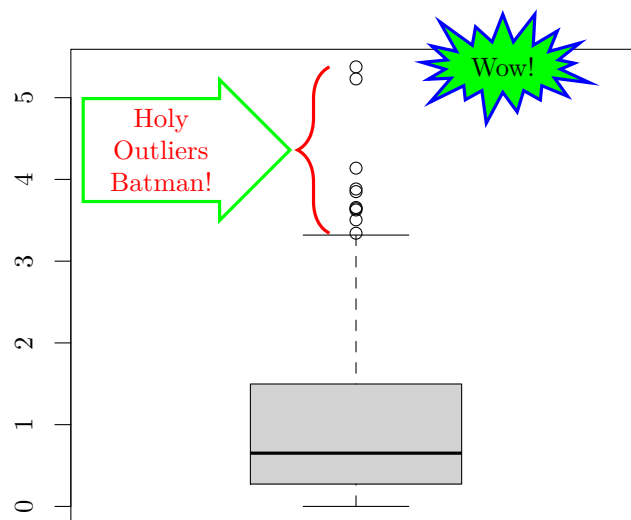


Figure 7: An example using *TikZ* annotation.

**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.

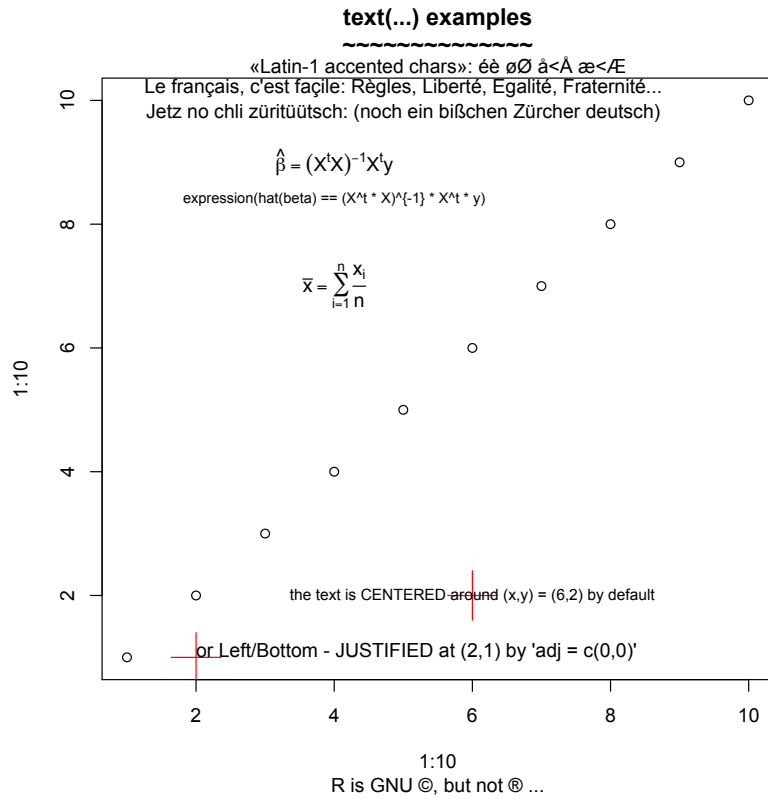


Figure 8: example(text) using the standard pdf() device.

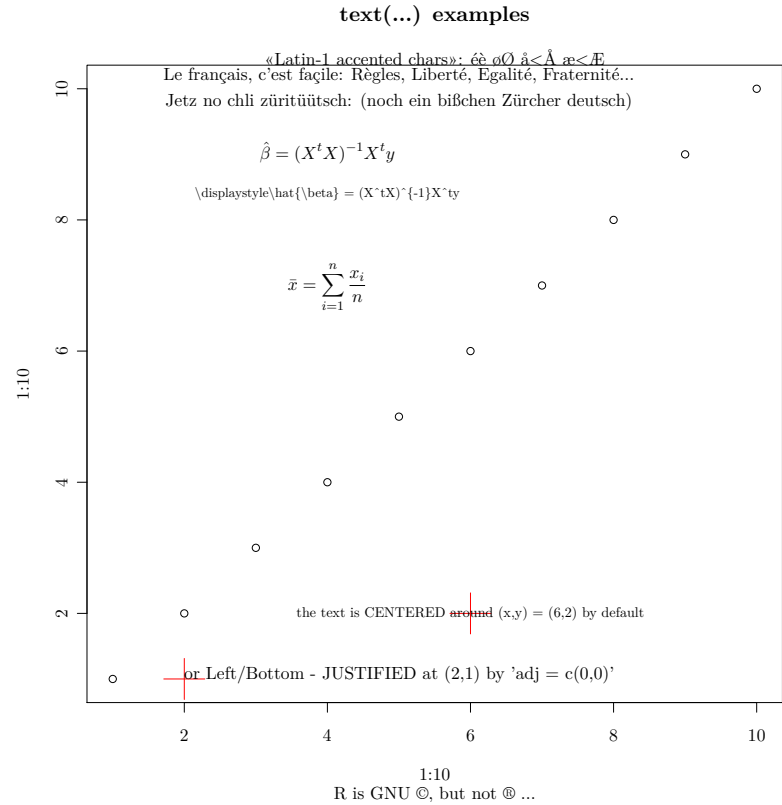


Figure 9: example(text) using tikz(..., engine = 'xetex').

## The getLatexCharMetrics and getLatexStrWidth Functions

### 4.1 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, a  $\LaTeX$  compiler will be invoked to generate them.

### 4.2 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.

`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.

### 4.3 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 ‘ $\alpha$ ’ 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. This is the distance between the baseline and the highest point of the character’s glyph.
- The descent of the character. This is 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')
```

```
Warning in getLatexCharMetrics("y"): getLatexCharMetrics only accepts integers!
```

```
NULL
```

```
getLatexCharMetrics(20)
```

```
Warning in getLatexCharMetrics(20): pdftex can only generate metrics for character codes between 32
and 126! See the Unicode section of ?tikzDevice for more information.
```

```
NULL
```

```
# Will return metrics for 'y'  
getLatexCharMetrics(121.99)
```

```
[1] 4.30450 1.94397 5.27649
```

# **Part II**

## **Installation Guide**

## Obtaining a L<sup>A</sup>T<sub>E</sub>X Distribution

This section offers pointers on how to obtain a L<sup>A</sup>T<sub>E</sub>X 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 provide integrated package managers which greatly simplify the process of installing additional L<sup>A</sup>T<sub>E</sub>X packages. Currently this section is not, and may never be, a troubleshooting guide for L<sup>A</sup>T<sub>E</sub>X installation. For those unfortunate situations we refer the user to the documentation of each distribution.

A L<sup>A</sup>T<sub>E</sub>X distribution provides the packages and support programs required by the **tikzDevice** and the documents that use its output. In addition to a L<sup>A</sup>T<sub>E</sub>X compiler, a few extension packages are required. [Section 5.4](#) describes how to obtain and install these packages.

### 5.1 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 L<sup>A</sup>T<sub>E</sub>X 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.

### 5.2 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. 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. Note that the version of TeX Live provided by many Linux package management systems sometimes lags behind the version provided directly by the TeX Users Group.

### 5.3 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 2013 this amounts to approximately 2.3 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.

### 5.4 Installing TikZ and Other Packages

Unsurprisingly, **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 `PGF` and both are distributed as the `pgf` package. Users who do not have a full L<sup>A</sup>T<sub>E</sub>X installation will also need to install a few more required packages:

`pgf` As mentioned, provides `TikZ`.

**preview** Used to crop documents in order to produce standalone figures.

**ms** Martin Schröder's LaTeX packages. `everyshi.sty` lets us run commands at every shipped page.

**graphics** L<sup>A</sup>T<sub>E</sub>X's general-purpose graphics inclusion functionality.

**pdftex-def** Device-specific colour and graphics definitions when running pdfL<sup>A</sup>T<sub>E</sub>X/pdfL<sup>A</sup>T<sub>E</sub>X.

**oberdiek** `infwarerr.sty` provides info/error/warning messages

**ec** (Font metrics for) the default font, European Computer Modern.

**xcolor** Used by TikZ to specify colors.

**fontspec** Used by LuaL<sup>A</sup>T<sub>E</sub>X and XeL<sup>A</sup>T<sub>E</sub>X to select fonts.

**xunicode** Assists LuaL<sup>A</sup>T<sub>E</sub>X and XeL<sup>A</sup>T<sub>E</sub>X with UTF-8 characters.

## Using a L<sup>A</sup>T<sub>E</sub>X Package Manager

The easiest way to install L<sup>A</sup>T<sub>E</sub>X 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 automatically 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
```

## 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 L<sup>A</sup>T<sub>E</sub>X 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 L<sup>A</sup>T<sub>E</sub>X package available.

Generally, the best place to find L<sup>A</sup>T<sub>E</sub>X packages is the Comprehensive TeX Archive Network, or CTAN located at <http://www.ctan.org>. In the case of the PGF/TikZ 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 L<sup>A</sup>T<sub>E</sub>X packages are stored in a specially directory called a `texmf` folder. Most T<sub>E</sub>X 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 an 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 T<sub>E</sub>X 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-compliant configuration. TDS-compliant archives usually have the phrase `tds` somewhere in their filename and may be installed from a UNIX shell<sup>1</sup> like so:

Installing L<sup>A</sup>T<sub>E</sub>X 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 L<sup>A</sup>T<sub>E</sub>X 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 L<sup>A</sup>T<sub>E</sub>X compiler to take notice of the additions. This is done with the `mktexlsr` command:

Registering new L<sup>A</sup>T<sub>E</sub>X 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
```

---

<sup>1</sup>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

### R User Types

```
tikz('Rplot.tex')  
plot(1)  
dev.off()
```

### TikZ Device Translates

```
static void TikZ_Circle( double x, double y, double r,  
    const pGEcontext plotParams, pDevDesc deviceInfo){  
  
    ...  
  
    printOutput(tikzInfo, "\n\\path["");  
    TikZ_WriteDrawOptions(plotParams, deviceInfo, ops);  
  
    /* End options, print coordinates. */  
    printOutput(tikzInfo, " ]_(%6.2f,%6.2f)_circle_(%6.2f);\n",  
        x,y,r);  
}
```

### L<sup>A</sup>T<sub>E</sub>X Output is Produced

```
% Created by tikzDevice  
% !TEX encoding = UTF-8 Unicode  
\begin{tikzpicture}[x=1pt,y=1pt]  
  
    ...  
  
    \path[draw=drawColor,line width= 0.4pt,  
        line join=round,line cap=round]  
        (264.94,258.95) circle ( 2.25);  
  
    ...  
  
\end{tikzpicture}
```

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

*Programming Perl*  
—LARRY WALL

CHAPTER 6

## Introduction and Background

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 as 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, L<sup>A</sup>T<sub>E</sub>X, 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*

CHAPTER 7

## 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:

### 7.1 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.



- `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.

## 7.2 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.

## 7.3 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.

CHAPTER 8

# 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 “g” printed to the left, the ascent has been calculated as: 24.99963pt

### 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 “g” printed to the left, the width has been calculated as: 20.0pt

Providing font metrics and string widths is without a doubt the most difficult task a R graphics device must undertake. The calculation of string widths is made even more 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:  $\alpha$ .

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?

## 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 itself. 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, etc. 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 L<sup>A</sup>T<sub>E</sub>X compiler to compute the numbers for us.

Recovering these numbers from the L<sup>A</sup>T<sub>E</sub>X compiler involves the execution of three additional tasks:

1. We must write a L<sup>A</sup>T<sub>E</sub>X input file that contains instructions for calculating the metrics.
2. We call the L<sup>A</sup>T<sub>E</sub>X 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 L<sup>A</sup>T<sub>E</sub>X. 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.

## 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 them 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.<sup>2</sup> 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.

## Implementing a System Call to L<sup>A</sup>T<sub>E</sub>X

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 L<sup>A</sup>T<sub>E</sub>X input:

#### Creating a L<sup>A</sup>T<sub>E</sub>X 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...
```

<sup>2</sup>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 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  $\LaTeX$  file.

#### 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  $\TeX$  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, 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: '!'.<sup>1</sup>

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 not 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`.

#### Extracting character dimensions using `TikZ`

```
\begin{tikzpicture}

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

\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} };

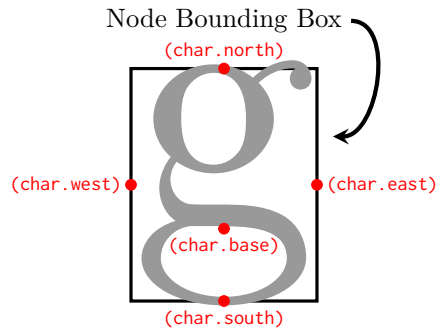
\path let \p1 = ($(char.base) - (char.south)$),
```

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

What the heck just happened? Well, first we instructed  $\text{\LaTeX}$  to enter the  $\text{\tikZ}$  picture environment using  $\text{\begin{tikzpicture}}$ . Then we ordered  $\text{\tikZ}$  to create a node named “char” containing the command  $\text{\char}$  followed by the value of  $\text{charCode}$ . 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  $\text{inner sep}$  and  $\text{outer sep}$  options are set to  $\text{0pt}$  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  $\text{\tikZ}$  defines ‘anchors’ along the bounding box of the node. All anchors are referred using a node name.position notation. Since we named the node  $\text{char}$ , all the anchors start with  $\text{char}$ . The anchor positions 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  $\text{char.north}$  and  $\text{char.base}$ . Similarly, for the descent we would calculate the difference in height between  $\text{char.base}$  and  $\text{char.south}$  and width can be obtained using  $\text{char.west}$  and  $\text{char.east}$ . This is the purpose of the admittedly cryptic  $\text{\path}$  commands that are inserted in the  $\text{\LaTeX}$  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  $\text{\path}$  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  $\text{\tikZ}$  code produced by  $\text{getLatexCharMetrics}$ , the  $\text{let}$  operation is specified. Basically,  $\text{let}$  postpones the actual drawing of a path and performs calculations until the  $\text{in}$  keyword is encountered. The result of these calculations are stored in a set of special variables which must start with  $\text{\n}$ ,  $\text{\p}$ ,  $\text{\x}$  or  $\text{\y}$ . The first  $\text{let}$  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 `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  $\text{\LaTeX}$  `.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  $\text{\LaTeX}$  compiler may be shut down, this is done by adding the final two lines to the input file:

Terminating a  $\text{\LaTeX}$  compilation

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

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

close(texIn)
```

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

Terminating a  $\text{\LaTeX}$  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  $\text{\LaTeX}$  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`.

CHAPTER 9

## 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!”

### 9.1 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.



## Bibliography

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*.

Urbanek, S. (2013), *png: Read and write PNG images*, r package version 0.1-7.