**Background**

Reading through the vignette Its spotted the image\_morph() function from Magick package. In this post , experiment with the function to build the GIF below that shows the changes in the England football first kit over time, using images from the excellent [Historical Football Kits](http://www.historicalkits.co.uk/) website.

Magick Package – A Brief Introduction

## **Installing magick**

On Windows or macOS the package is most easily installed via CRAN.

install.packages("magick")

The binary CRAN packages work out of the box and have most important features enabled. Use magick\_config to see which features and formats are supported by your version of ImageMagick.

**library**(magick)

## Linking to ImageMagick 6.9.12.3

## Enabled features: cairo, fontconfig, freetype, heic, lcms, pango, raw, rsvg, webp

## Disabled features: fftw, ghostscript, x11

str(magick::magick\_config())

## List of 24

## $ version :Class 'numeric\_version' hidden list of 1

## ..$ : int [1:4] 6 9 12 3

## $ modules : logi FALSE

## $ cairo : logi TRUE

## $ fontconfig : logi TRUE

## $ freetype : logi TRUE

## $ fftw : logi FALSE

## $ ghostscript : logi FALSE

## $ heic : logi TRUE

## $ jpeg : logi TRUE

## $ lcms : logi TRUE

## $ libopenjp2 : logi TRUE

## $ lzma : logi TRUE

## $ pangocairo : logi TRUE

## $ pango : logi TRUE

## $ png : logi TRUE

## $ raw : logi TRUE

## $ rsvg : logi TRUE

## $ tiff : logi TRUE

## $ webp : logi TRUE

## $ wmf : logi FALSE

## $ x11 : logi FALSE

## $ xml : logi TRUE

## $ zero-configuration: logi TRUE

## $ threads : int 1

### **Build from source**

On Linux you need to install the ImageMagick++ library: on Debian/Ubuntu this is called libmagick++-dev:

sudo apt-get install libmagick++-dev

On Fedora or CentOS/RHEL we need ImageMagick-c++-devel:

sudo yum install ImageMagick-c++-devel

To install from source on macOS you need imagemagick@6 from homebrew.

brew install imagemagick@6

Unfortunately the current imagemagick@6 configuration on homebrew disables a bunch of features, including librsvg and fontconfig. Therefore the quality of fonts and svg rendering might be suboptimal. The is not a problem for the CRAN binary package.

## **Image IO**

What makes magick so magical is that it automatically converts and renders all common image formats. ImageMagick supports dozens of formats and automatically detects the type. Use magick::magick\_config() to list the formats that your version of ImageMagick supports.

### **Read and write**

Images can be read directly from a file path, URL, or raw vector with image data with image\_read. The image\_info function shows some meta data about the image, similar to the imagemagick identify command line utility.

**library**(magick)

tiger <- image\_read\_svg('http://jeroen.github.io/images/tiger.svg', width = 350)

print(tiger)

## format width height colorspace matte filesize density

## 1 PNG 350 350 sRGB TRUE 0 72x72



We use image\_write to export an image in any format to a file on disk, or in memory if path = NULL.

*# Render svg to png bitmap*

image\_write(tiger, path = "tiger.png", format = "png")

If path is a filename, image\_write returns path on success such that the result can be piped into function taking a file path.

### **Converting formats**

Magick keeps the image in memory in its original format. Specify the format parameter image\_write to convert to another format. You can also internally convert the image to another format earlier, before applying transformations. This can be useful if your original format is lossy.

tiger\_png <- image\_convert(tiger, "png")

image\_info(tiger\_png)

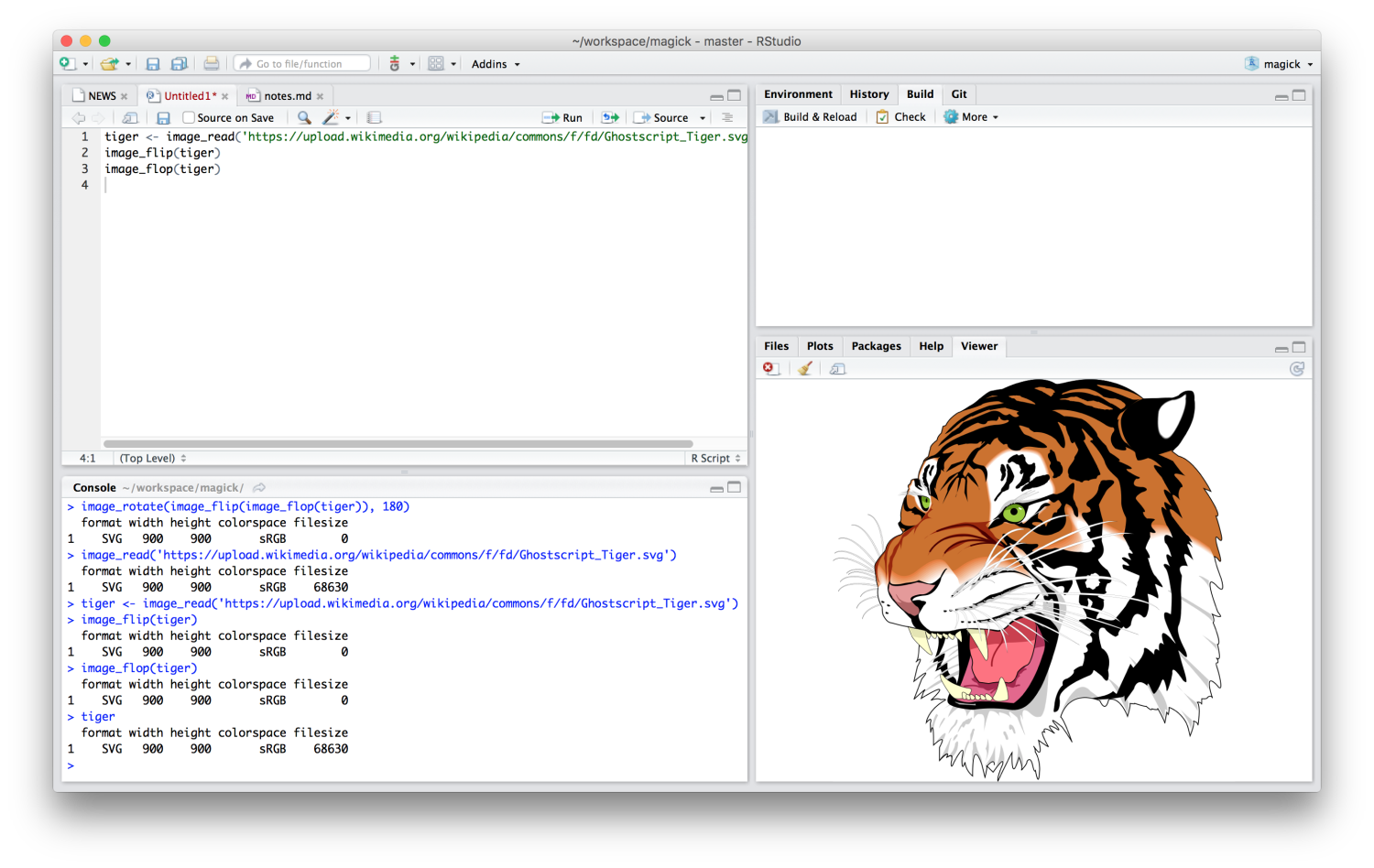
## format width height colorspace matte filesize density

## 1 PNG 350 350 sRGB TRUE 0 72x72

Note that size is currently 0 because ImageMagick is lazy (in the good sense) and does not render until it has to.

### **Preview**

IDE’s with a built-in web browser (such as RStudio) automatically display magick images in the viewer. This results in a neat interactive image editing environment.



Alternatively, on Linux you can use image\_display to preview the image in an X11 window. Finally image\_browse opens the image in your system’s default application for a given type.

*# X11 only*

image\_display(tiger)

*# System dependent*

image\_browse(tiger)

Another method is converting the image to a raster object and plot it on R’s graphics display. However this is very slow and only useful in combination with other plotting functionality. See #raster below.

## **Transformations**

The best way to get a sense of available transformations is walk through the examples in the ?transformations help page in RStudio. Below a few examples to get a sense of what is possible.

### **Cut and edit**

Several of the transformation functions take an geometry parameter which requires a special syntax of the form AxB+C+D where each element is optional. Some examples:

* image\_crop(image, "100x150+50"): crop out *width:100px* and *height:150px* starting *+50px* from the left
* image\_scale(image, "200"): resize proportionally to width: *200px*
* image\_scale(image, "x200"): resize proportionally to height: *200px*
* image\_fill(image, "blue", "+100+200"): flood fill with blue starting at the point at *x:100, y:200*
* image\_border(frink, "red", "20x10"): adds a border of 20px left+right and 10px top+bottom

The full syntax is specified in the Magick::Geometry documentation.

*# Example image*

frink <- image\_read("https://jeroen.github.io/images/frink.png")

print(frink)

## format width height colorspace matte filesize density

## 1 PNG 220 445 sRGB TRUE 73494 72x72



*# Add 20px left/right and 10px top/bottom*

image\_border(image\_background(frink, "hotpink"), "#000080", "20x10")



*# Trim margins*

image\_trim(frink)



*# Passport pica*

image\_crop(frink, "100x150+50")



*# Resize*

image\_scale(frink, "300") *# width: 300px*

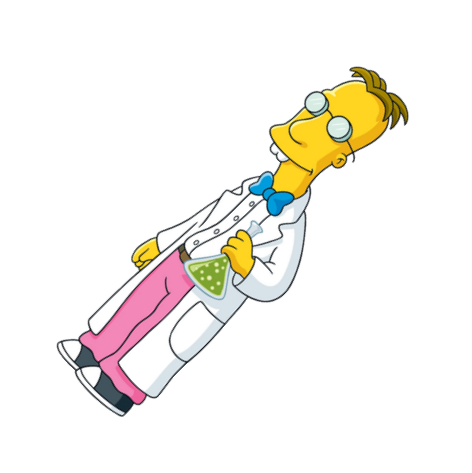


image\_scale(frink, "x300") *# height: 300px*

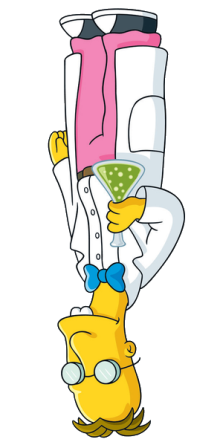


*# Rotate or mirror*

image\_rotate(frink, 45)



image\_flip(frink)



image\_flop(frink)



*# Brightness, Saturation, Hue*

image\_modulate(frink, brightness = 80, saturation = 120, hue = 90)



*# Paint the shirt orange*

image\_fill(frink, "orange", point = "+100+200", fuzz = 20)



With image\_fill we can flood fill starting at pixel point. The fuzz parameter allows for the fill to cross for adjacent pixels with similarish colors. Its value must be between 0 and 256^2 specifying the max geometric distance between colors to be considered equal. Here we give professor frink an orange shirt for the World Cup.

### **Filters and effects**

ImageMagick also has a bunch of standard effects that are worth checking out.

*# Add randomness*

image\_blur(frink, 10, 5)

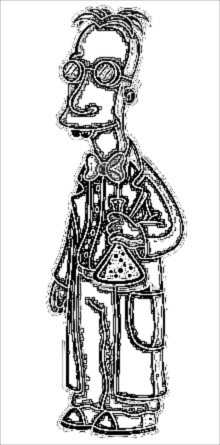


image\_noise(frink)



*# Silly filters*

image\_charcoal(frink)



image\_oilpaint(frink)



image\_negate(frink)



### **Kernel convolution**

The image\_convolve() function applies a kernel over the image. Kernel convolution means that each pixel value is recalculated using the weighted neighborhood sum defined in the kernel matrix. For example lets look at this simple kernel:

kern <- matrix(0, ncol = 3, nrow = 3)

kern[1, 2] <- 0.25

kern[2, c(1, 3)] <- 0.25

kern[3, 2] <- 0.25

kern

## [,1] [,2] [,3]

## [1,] 0.00 0.25 0.00

## [2,] 0.25 0.00 0.25

## [3,] 0.00 0.25 0.00

This kernel changes each pixel to the mean of its horizontal and vertical neighboring pixels, which results in a slight blurring effect in the right-hand image below:

img <- image\_resize(logo, "300x300")

img\_blurred <- image\_convolve(img, kern)

image\_append(c(img, img\_blurred))



Or use any of the standard kernels

img %>% image\_convolve('Sobel') %>% image\_negate()



img %>% image\_convolve('DoG:0,0,2') %>% image\_negate()



### **Text annotation**

Finally it can be useful to print some text on top of images:

*# Add some text*

image\_annotate(frink, "I like R!", size = 70, gravity = "southwest", color = "green")



*# Customize text*

image\_annotate(frink, "CONFIDENTIAL", size = 30, color = "red", boxcolor = "pink",

degrees = 60, location = "+50+100")



*# Fonts may require ImageMagick has fontconfig*

image\_annotate(frink, "The quick brown fox", font = 'Times', size = 30)



Fonts that are supported on most platforms include "sans", "mono", "serif", "Times", "Helvetica", "Trebuchet", "Georgia", "Palatino"or "Comic Sans".

### **Combining with pipes**

Each of the image transformation functions returns a **modified copy** of the original image. It does not affect the original image.

frink <- image\_read("https://jeroen.github.io/images/frink.png")

frink2 <- image\_scale(frink, "100")

image\_info(frink)

## format width height colorspace matte filesize density

## 1 PNG 220 445 sRGB TRUE 73494 72x72

image\_info(frink2)

## format width height colorspace matte filesize density

## 1 PNG 100 202 sRGB TRUE 0 72x72

Hence to combine transformations you need to chain them:

test <- image\_rotate(frink, 90)

test <- image\_background(test, "blue", flatten = TRUE)

test <- image\_border(test, "red", "10x10")

test <- image\_annotate(test, "This is how we combine transformations", color = "white", size = 30)

print(test)

## format width height colorspace matte filesize density

## 1 PNG 465 240 sRGB TRUE 0 72x72



Using magrittr pipe syntax makes it a bit more readable

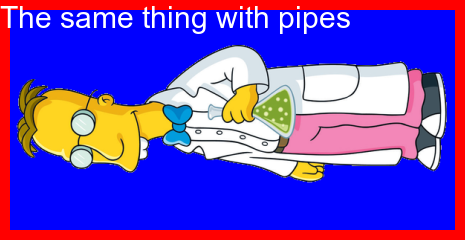
image\_read("https://jeroen.github.io/images/frink.png") %>%

image\_rotate(270) %>%

image\_background("blue", flatten = TRUE) %>%

image\_border("red", "10x10") %>%

image\_annotate("The same thing with pipes", color = "white", size = 30)



## **Image Vectors**

The examples above concern single images. However all functions in magick have been vectorized to support working with layers, compositions or animation.

The standard base methods [ [[, c() and length() are used to manipulate vectors of images which can then be treated as layers or frames.

*# Download earth gif and make it a bit smaller for vignette*

earth <- image\_read("https://jeroen.github.io/images/earth.gif") %>%

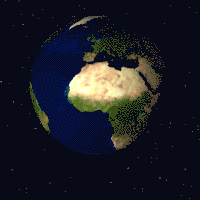
image\_scale("200x") %>%

image\_quantize(128)

length(earth)

## [1] 44

earth



head(image\_info(earth))

## format width height colorspace matte filesize density

## 1 GIF 200 200 RGB FALSE 0 72x72

## 2 GIF 200 200 RGB TRUE 0 72x72

## 3 GIF 200 200 RGB TRUE 0 72x72

## 4 GIF 200 200 RGB TRUE 0 72x72

## 5 GIF 200 200 RGB TRUE 0 72x72

## 6 GIF 200 200 RGB TRUE 0 72x72

rev(earth) %>%

image\_flip() %>%

image\_annotate("meanwhile in Australia", size = 20, color = "white")



### **Layers**

We can stack layers on top of each other as we would in Photoshop:

bigdata <- image\_read('https://jeroen.github.io/images/bigdata.jpg')

frink <- image\_read("https://jeroen.github.io/images/frink.png")

logo <- image\_read("https://jeroen.github.io/images/Rlogo.png")

img <- c(bigdata, logo, frink)

img <- image\_scale(img, "300x300")

image\_info(img)

## format width height colorspace matte filesize density

## 1 JPEG 300 225 sRGB FALSE 0 72x72

## 2 PNG 300 232 sRGB TRUE 0 72x72

## 3 PNG 148 300 sRGB TRUE 0 72x72

A mosaic prints images on top of one another, expanding the output canvas such that that everything fits:

image\_mosaic(img)



Flattening combines the layers into a single image which has the size of the first image:

image\_flatten(img)

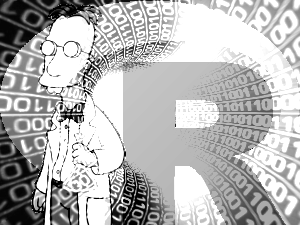


Flattening and mosaic allow for specifying alternative composite operators:

image\_flatten(img, 'Add')



image\_flatten(img, 'Modulate')



image\_flatten(img, 'Minus')



### **Combining**

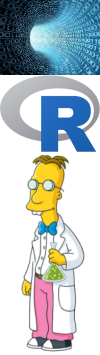
Appending means simply putting the frames next to each other:

image\_append(image\_scale(img, "x200"))



Use stack = TRUE to position them on top of each other:

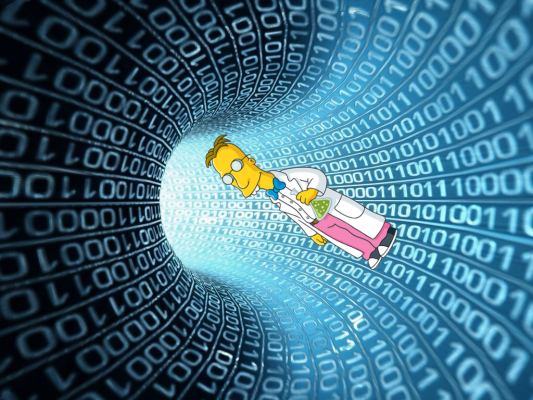
image\_append(image\_scale(img, "100"), stack = TRUE)



Composing allows for combining two images on a specific position:

bigdatafrink <- image\_scale(image\_rotate(image\_background(frink, "none"), 300), "x200")

image\_composite(image\_scale(bigdata, "x400"), bigdatafrink, offset = "+180+100")



### **Pages**

When reading a PDF document, each page becomes an element of the vector. Note that PDF gets rendered while reading so you need to specify the density immediately.

manual <- image\_read\_pdf('https://cloud.r-project.org/web/packages/magick/magick.pdf', density = 72)

image\_info(manual)

## format width height colorspace matte filesize density

## 1 PNG 612 792 sRGB TRUE 0 72x72

## 2 PNG 612 792 sRGB TRUE 0 72x72

## 3 PNG 612 792 sRGB TRUE 0 72x72

## 4 PNG 612 792 sRGB TRUE 0 72x72

## 5 PNG 612 792 sRGB TRUE 0 72x72

## 6 PNG 612 792 sRGB TRUE 0 72x72

## 7 PNG 612 792 sRGB TRUE 0 72x72

## 8 PNG 612 792 sRGB TRUE 0 72x72

## 9 PNG 612 792 sRGB TRUE 0 72x72

## 10 PNG 612 792 sRGB TRUE 0 72x72

## 11 PNG 612 792 sRGB TRUE 0 72x72

## 12 PNG 612 792 sRGB TRUE 0 72x72

## 13 PNG 612 792 sRGB TRUE 0 72x72

## 14 PNG 612 792 sRGB TRUE 0 72x72

## 15 PNG 612 792 sRGB TRUE 0 72x72

## 16 PNG 612 792 sRGB TRUE 0 72x72

## 17 PNG 612 792 sRGB TRUE 0 72x72

## 18 PNG 612 792 sRGB TRUE 0 72x72

## 19 PNG 612 792 sRGB TRUE 0 72x72

## 20 PNG 612 792 sRGB TRUE 0 72x72

## 21 PNG 612 792 sRGB TRUE 0 72x72

## 22 PNG 612 792 sRGB TRUE 0 72x72

## 23 PNG 612 792 sRGB TRUE 0 72x72

## 24 PNG 612 792 sRGB TRUE 0 72x72

## 25 PNG 612 792 sRGB TRUE 0 72x72

## 26 PNG 612 792 sRGB TRUE 0 72x72

## 27 PNG 612 792 sRGB TRUE 0 72x72

## 28 PNG 612 792 sRGB TRUE 0 72x72

## 29 PNG 612 792 sRGB TRUE 0 72x72

## 30 PNG 612 792 sRGB TRUE 0 72x72

## 31 PNG 612 792 sRGB TRUE 0 72x72

## 32 PNG 612 792 sRGB TRUE 0 72x72

## 33 PNG 612 792 sRGB TRUE 0 72x72

## 34 PNG 612 792 sRGB TRUE 0 72x72

## 35 PNG 612 792 sRGB TRUE 0 72x72

## 36 PNG 612 792 sRGB TRUE 0 72x72

## 37 PNG 612 792 sRGB TRUE 0 72x72

## 38 PNG 612 792 sRGB TRUE 0 72x72

## 39 PNG 612 792 sRGB TRUE 0 72x72

## 40 PNG 612 792 sRGB TRUE 0 72x72

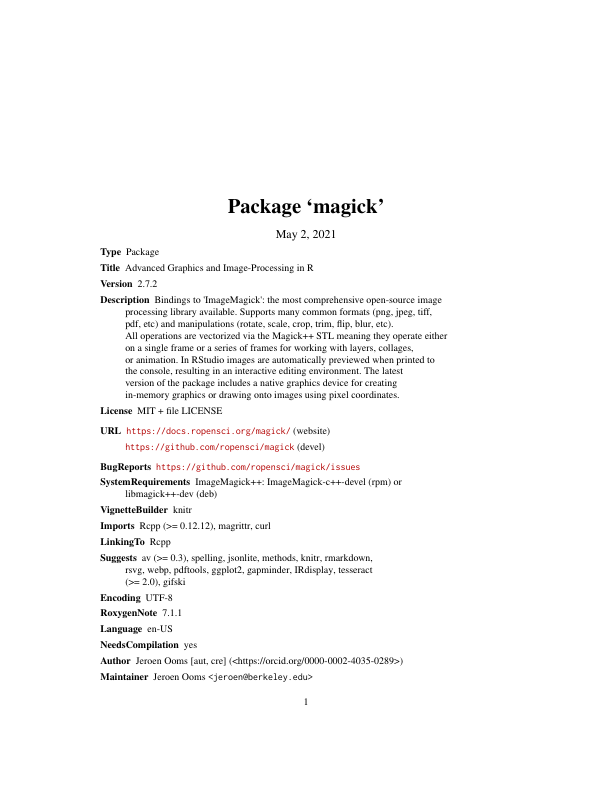
## 41 PNG 612 792 sRGB TRUE 0 72x72

## 42 PNG 612 792 sRGB TRUE 0 72x72

## 43 PNG 612 792 sRGB TRUE 0 72x72

## 44 PNG 612 792 sRGB TRUE 0 72x72

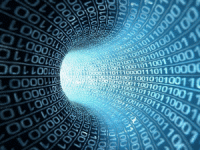
manual[1]



### **Animation**

Instead of treating vector elements as layers, we can also make them frames in an animation!

image\_animate(image\_scale(img, "200x200"), fps = 1, dispose = "previous")



Morphing creates a sequence of n images that gradually morph one image into another. It makes animations

newlogo <- image\_scale(image\_read("https://jeroen.github.io/images/Rlogo.png"))

oldlogo <- image\_scale(image\_read("https://jeroen.github.io/images/Rlogo-old.png"))

image\_resize(c(oldlogo, newlogo), '200x150!') %>%

image\_background('white') %>%

image\_morph() %>%

image\_animate(optimize = TRUE)



If you read in an existing GIF or Video file, each frame becomes a layer:

*# Foreground image*

banana <- image\_read("https://jeroen.github.io/images/banana.gif")

banana <- image\_scale(banana, "150")

image\_info(banana)

## format width height colorspace matte filesize density

## 1 GIF 150 148 sRGB TRUE 0 72x72

## 2 GIF 150 148 sRGB TRUE 0 72x72

## 3 GIF 150 148 sRGB TRUE 0 72x72

## 4 GIF 150 148 sRGB TRUE 0 72x72

## 5 GIF 150 148 sRGB TRUE 0 72x72

## 6 GIF 150 148 sRGB TRUE 0 72x72

## 7 GIF 150 148 sRGB TRUE 0 72x72

## 8 GIF 150 148 sRGB TRUE 0 72x72

Manipulate the individual frames and put them back into an animation:

*# Background image*

background <- image\_background(image\_scale(logo, "200"), "white", flatten = TRUE)

*# Combine and flatten frames*

frames <- image\_composite(background, banana, offset = "+70+30")

*# Turn frames into animation*

animation <- image\_animate(frames, fps = 10, optimize = TRUE)

print(animation)

## format width height colorspace matte filesize density

## 1 gif 200 155 sRGB TRUE 0 72x72

## 2 gif 94 105 sRGB TRUE 0 72x72

## 3 gif 125 122 sRGB TRUE 0 72x72

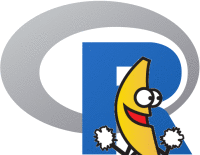
## 4 gif 108 118 sRGB TRUE 0 72x72

## 5 gif 108 105 sRGB TRUE 0 72x72

## 6 gif 92 105 sRGB TRUE 0 72x72

## 7 gif 113 123 sRGB TRUE 0 72x72

## 8 gif 119 118 sRGB TRUE 0 72x72



Animations can be saved as GIF of MPEG files:

image\_write(animation, "Rlogo-banana.gif")

## **Drawing and Graphics**

A relatively recent addition to the package is a native R graphics device which produces a magick image object. This can either be used like a regular device for making plots, or alternatively to open a device which draws onto an existing image using pixel coordinates.

### **Graphics device**

The image\_graph() function opens a new graphics device similar to e.g. png() or x11(). It returns an image object to which the plot(s) will be written. Each “page” in the plotting device will become a frame in the image object.

*# Produce image using graphics device*

fig <- image\_graph(width = 400, height = 400, res = 96)

ggplot2::qplot(mpg, wt, data = mtcars, colour = cyl)

dev.off()

We can easily post-process the figure using regular image operations.

*# Combine*

out <- image\_composite(fig, frink, offset = "+70+30")

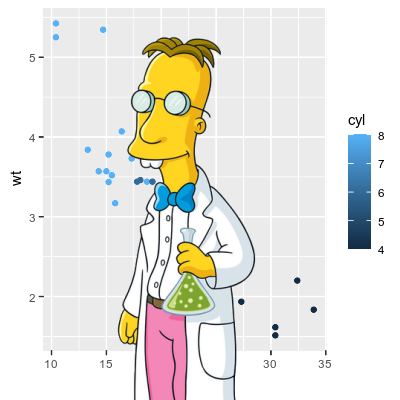
print(out)

## # A tibble: 1 × 7

## format width height colorspace matte filesize density

## <chr> <int> <int> <chr> <lgl> <int> <chr>

## 1 PNG 400 400 sRGB TRUE 0 96x96



### **Drawing device**

Another way to use the graphics device is to draw on top of an exiting image using pixel coordinates.

*# Or paint over an existing image*

img <- image\_draw(frink)

rect(20, 20, 200, 100, border = "red", lty = "dashed", lwd = 5)

abline(h = 300, col = 'blue', lwd = '10', lty = "dotted")

text(30, 250, "Hoiven-Glaven", family = "monospace", cex = 4, srt = 90)

palette(rainbow(11, end = 0.9))

symbols(rep(200, 11), seq(0, 400, 40), circles = runif(11, 5, 35),

bg = 1:11, inches = FALSE, add = TRUE)

dev.off()

print(img)

## # A tibble: 1 × 7

## format width height colorspace matte filesize density

## <chr> <int> <int> <chr> <lgl> <int> <chr>

## 1 PNG 220 445 sRGB TRUE 0 72x72



By default image\_draw() sets all margins to 0 and uses graphics coordinates to match image size in pixels (width x height) where (0,0) is the top left corner. Note that this means the y axis increases from top to bottom which is the opposite of typical graphics coordinates. You can override all this by passing custom xlim, ylim or mar values to image\_draw.

### **Animated Graphics**

The graphics device supports multiple frames which makes it easy to create animated graphics. The code below shows how you would implement the example from the very cool gganimate package using the magick graphics device.

**library**(gapminder)

**library**(ggplot2)

img <- image\_graph(600, 340, res = 96)

datalist <- split(gapminder, gapminder$year)

out <- lapply(datalist, **function**(data){

p <- ggplot(data, aes(gdpPercap, lifeExp, size = pop, color = continent)) +

scale\_size("population", limits = range(gapminder$pop)) + geom\_point() + ylim(20, 90) +

scale\_x\_log10(limits = range(gapminder$gdpPercap)) + ggtitle(data$year) + theme\_classic()

print(p)

})

dev.off()

animation <- image\_animate(img, fps = 2, optimize = TRUE)

print(animation)

## # A tibble: 12 × 7

## format width height colorspace matte filesize density

## <chr> <int> <int> <chr> <lgl> <int> <chr>

## 1 gif 600 340 sRGB TRUE 0 96x96

## 2 gif 385 243 sRGB TRUE 0 96x96

## 3 gif 395 237 sRGB TRUE 0 96x96

## 4 gif 374 232 sRGB TRUE 0 96x96

## 5 gif 393 225 sRGB TRUE 0 96x96

## 6 gif 373 234 sRGB TRUE 0 96x96

## 7 gif 354 234 sRGB TRUE 0 96x96

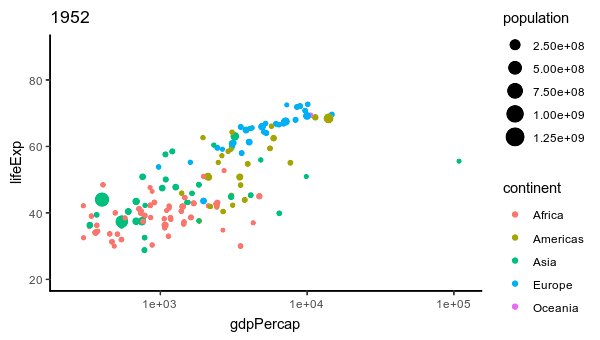
## 8 gif 308 210 sRGB TRUE 0 96x96

## 9 gif 320 260 sRGB TRUE 0 96x96

## 10 gif 331 218 sRGB TRUE 0 96x96

## 11 gif 356 208 sRGB TRUE 0 96x96

## 12 gif 347 208 sRGB TRUE 0 96x96



To write it to a file you would simply do:

image\_write(animation, "gapminder.gif")

## **Raster Images**

Magick images can also be converted to raster objects for use with R’s graphics device. Thereby we can combine it with other graphics tools. However do note that R’s graphics device is very slow and has a very different coordinate system which reduces the quality of the image.

### **Base R rasters**

Base R has an as.raster format which converts the image to a vector of strings. The paper Raster Images in R Graphics by Paul Murrell gives a nice overview.

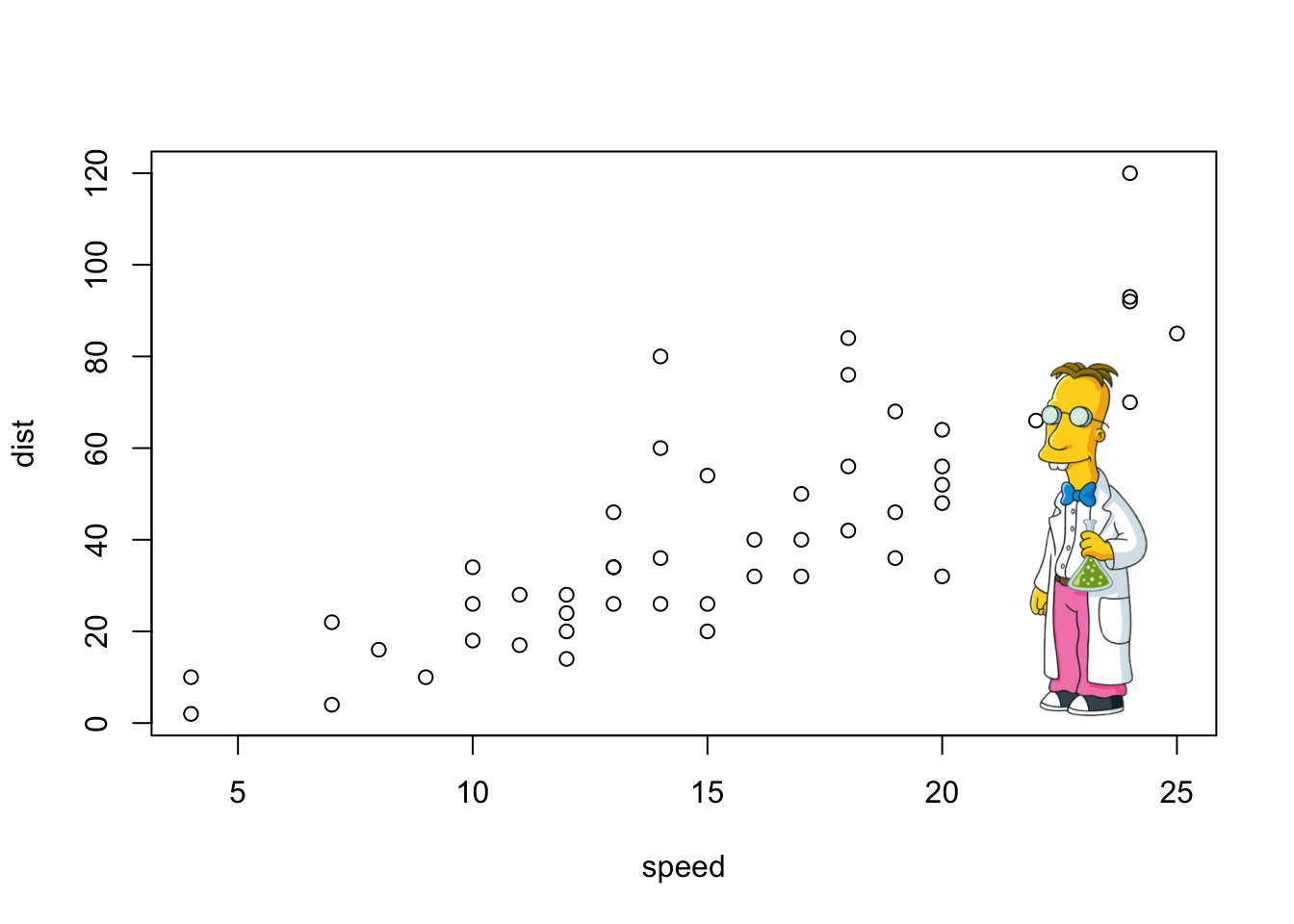
plot(as.raster(frink))



*# Print over another graphic*

plot(cars)

rasterImage(frink, 21, 0, 25, 80)



### **The grid package**

The grid package makes it easier to overlay a raster on the graphics device without having to adjust for the x/y coordinates of the plot.

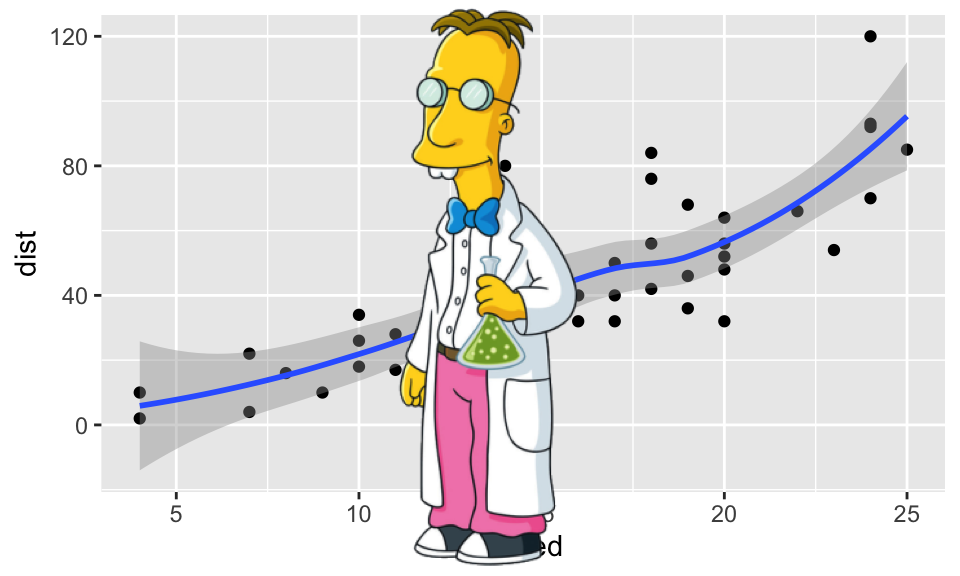
**library**(ggplot2)

**library**(grid)

qplot(speed, dist, data = cars, geom = c("point", "smooth"))

## `geom\_smooth()` using method = 'loess' and formula 'y ~ x'

grid.raster(frink)



## **OCR text extraction**

A recent addition to the package is to extract text from images using OCR. This requires the tesseract package:

install.packages("tesseract")

img <- image\_read("http://jeroen.github.io/images/testocr.png")

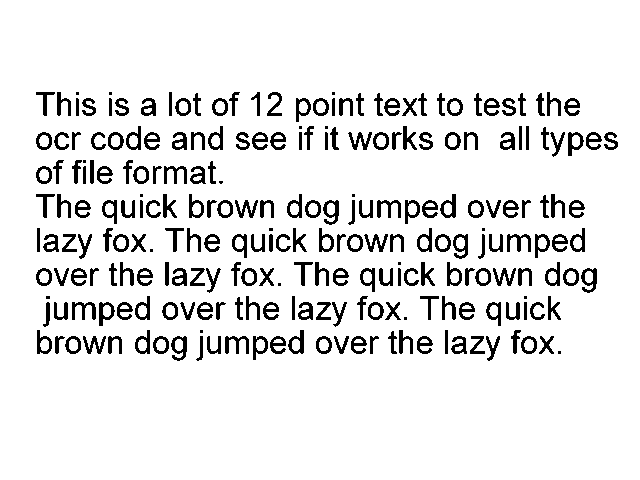
print(img)

## # A tibble: 1 × 7

## format width height colorspace matte filesize density

## <chr> <int> <int> <chr> <lgl> <int> <chr>

## 1 PNG 640 480 sRGB TRUE 23359 72x72



*# Extract text*

cat(image\_ocr(img))

## This is a lot of 12 point text to test the

## ocr code and see if it works on all types

## of file format.

##

## The quick brown dog jumped over the

## lazy fox. The quick brown dog jumped

## over the lazy fox. The quick brown dog

## jumped over the lazy fox. The quick

## brown dog jumped over the lazy fox.

**Scraping**

The Historical Football Kits website has a detailed section on England kits spread over six pages, starting from the first outfits used in [1872](http://www.historicalkits.co.uk/international/england/england-1872-1939.html). Each pages includes some interesting discussion – and importantly for this post – images of the kits.

We can use the read\_html() from the xml2 package and map() from purrr to read and save the source code of each page.

library(rvest)

library(tidyverse)

library(xml2)

library(purr)

htmls <- c(

"http://www.historicalkits.co.uk/international/england/england-1872-1939.html",

"http://www.historicalkits.co.uk/international/england/england-1946-1960.html",

"http://www.historicalkits.co.uk/international/england/england-1960-1983.html",

"http://www.historicalkits.co.uk/international/england/england-1984-1997.html",

"http://www.historicalkits.co.uk/international/england/england-1997-2010.html",

"http://www.historicalkits.co.uk/international/england/england-2010-2019.html"

) %>%

map(read\_html)

From the source code we can then find the URLs of each kit image files using html\_nodes() and html\_attr() from rvest. I used purrr’s map\_dfr() to store the links in a tibble and then dropped rows that do not contain kit image links or are images of away kits, kits used in single game or links to shops to buy replicas. This filtering was based on the image label or image URL and performed with the aid of the str\_detect() function from stringr.

library(stringr)

library(rvest)

scrape\_img\_url <- function(html){

html %>%

html\_nodes(".float p , .float img") %>%

html\_attr("src") %>%

tbl\_df() %>%

set\_names("img\_url") %>%

mutate(label = html %>%

html\_nodes(".float p , .float img") %>%

html\_text() %>%

c(., NA) %>%

.[-1])

}

d1 <- htmls %>%

map\_dfr(scrape\_img\_url) %>%

filter(str\_detect(string = img\_url, pattern = "/international/england"),

!str\_detect(string = label, pattern = "change|alternate|Alternate|Change"),

!str\_detect(string = label, pattern = " v |Third"),

!str\_detect(string = img\_url, pattern = "lithuania|italy|yellow|red"))

head(d1)

## img\_url label

## 1 /international/england/images/england-1872.gif 1872

## 2 /international/england/images/england-1882.gif 1879-1900

## 3 /international/england/images/england-1900.gif 1900-1914

## 4 /international/england/images/england-1920-1932.gif 1920-1930

## 5 /international/england/images/england-1921.gif 1930-1934

## 6 /international/england/images/england-1934.gif 1934

Given these URLs I then downloaded each of the images which are stored in a single R object kits

library(magick)

kits <- d1 %>%

mutate(img\_url = paste0("http://www.historicalkits.co.uk", img\_url),

img\_url = str\_replace(string =img\_url, pattern =" ", replacement = "%20")) %>%

select(img\_url) %>%

map(image\_read) %>%

set\_names("img")

Typing kits into R will display each kit in the RStudio viewer (it will quickly run through each image). The console displays summary information for each image in the kits object.

> kits

$img

format width height colorspace filesize

1 GIF 170 338 sRGB 0

2 GIF 170 338 sRGB 0

3 GIF 170 338 sRGB 0

4 GIF 170 338 sRGB 0

5 GIF 170 338 sRGB 0

6 GIF 170 338 sRGB 0

7 GIF 170 338 sRGB 0

8 GIF 170 338 sRGB 0

9 GIF 170 338 sRGB 0

10 GIF 170 338 sRGB 0

**Annotating Images**

Before creating any GIF I wanted add annotations for the year and the copyright information. To do this I first created a border using image\_border() in magick and then image\_annotate() to add the text. I wrapped these edits into an add\_text() function and then applied each to the kit images.

add\_text <- function(img, label){

img %>%

image\_border(geometry = "10x60", color = "white") %>%

image\_chop("0x45") %>%

image\_annotate(text = label, gravity = "north") %>%

image\_annotate(

text = "Animation by @guyabelguyabel", gravity = "south", location = "+0+45"

) %>%

image\_annotate(

text = "Images are Copyright of Historical\nFootball Kits and reproduced by\nkind permission.",

gravity = "south"

)

}

for(i in 1:length(kits$img)){

kits$img[i] <- add\_text(img = kits$img[i], label = d1$label[i])

}

**Creating a GIF**

The final step was to bind together the set of images in an animated GIF with smooth transition images between each frame. To do this I used the image\_morph() twice. First to repeat the same image so that the GIF would remain stable for a few frames (kits\_morph1 below). Then again to create a set of morphing images between successive kits (kits\_morph0 below). The full set of frames were stored in kits\_ani

kits\_ani <- image\_morph(c(kits$img[1], kits$img[1]), frames = 5)

for(i in 2:length(kits$img)){

kits\_morph0 <- image\_morph(c(kits$img[i-1], kits$img[i]), frames = 5)

kits\_morph1 <- image\_morph(c(kits$img[i], kits$img[i]), frames = 5)

kits\_ani <- c(kits\_ani, kits\_morph0)

kits\_ani <- c(kits\_ani, kits\_morph1)

}

To create an animation I passed the set of frames in the kits\_morph object to the image\_animate() and image\_write() functions to give the image above.

kits\_ani %>%

image\_animate(fps = 10) %>%

image\_write(path = "england.gif")

**Club Teams**

Similar code as above can be used to create images for club teams. I tried this out for the mighty Reading. As the Reading kits on [Historical Football Kits](http://www.historicalkits.co.uk/Reading/Reading.htm) are on only one page and includes only home kits, finding the image URLs was much easier…

d1 <- read\_html("http://www.historicalkits.co.uk/Reading/Reading.htm") %>%

scrape\_img\_url() %>%

filter(str\_detect(string = img\_url, pattern = "/Reading"),

!str\_detect(string = img\_url, pattern = "unknown")) %>%

mutate(

label = str\_replace\_all(string = label,

pattern = "[:alpha:]|\\s",

replacement = "")

)