Reproducible image analysis in R

Ken Locey

24 August, 2015

OVERVIEW

This .pdf was created by an R Markdown document (.Rmd) that brings together nicely formatted text, runnable R code, and data (yes, even photos) to coexist in a single document. Running the R Markdown file (i.e., RPP_Analysis.Rmd) in an up-to-date version of RStudio 'knits' text, code, figures int an attractively formatted .pdf. Here, I will demonstrate how toanalyze and edit an image (.jpg) in a highly reproducible way and scientifically analyzable way, using freely accessible open source R code.

Hooray for open, transparent, and reproducible science!

The following borrow heavily from others with R image analysis know-how: http://www.r-bloggers.com/r-image-analysis-using-ebimage/. For information on any function, just type help(FunctionName) in your RStudio console, where FunctionName is the name of the function you're curious about.

1. set our working directory in RStudio

```
# Retrieve and set the working directory
rm(list=ls())
getwd()
setwd("~/GitHub/RepeatPhotoPoints")
```

2. Load packages

Here's we'll use biocLite: http://www.bioconductor.org/packages/release/bioc/vignettes/EBImage/inst/doc/EBImage-introduction.pdf

```
# source("http://bioconductor.org/biocLite.R") # install if needed
#biocLite("EBImage")
require(EBImage)

# Can also use ripa: https://cran.r-project.org/web/packages/ripa/ripa.pdf
#install.packages("ripa") # install if needed
require(ripa)
```

3. Reading in a jpg with EBImage.

```
Image <- readImage('~/GitHub/RepeatPhotoPoints/RPP35/RPP35.jpg')

# You can display it in your internet browser
#display(Image)
```

```
# Or display it in your RStudio viewer
display(Image, method="raster")
```

4. **Resizing the image, in this case, to make it easier to quickly work with.**

```
Image.256 <- resize(Image, w=256)
display(Image.256, method = "raster")
```

5. **Interested in the jpg's data?**

```
print(Image.256)
```

6. Let's have a small demonstration of how easy it is to format images.

A. Adjusting Brightness

It is better to start with the basic first, one of which is the brightness. As discussed above, brightness can be manipulated using + or -

```
Image1 <- Image.256 + 0.2
Image2 <- Image.256 - 0.2

# Let's display them in our RStudio environment
display(Image1, method="raster")
```

```
display(Image2, method="raster")
```

B. Adjusting Contrast

Contrast can be manipulated using multiplication operator(*)

```
Image3 <- Image.256 * 0.5
Image4 <- Image.256 * 2

# Let's display them in our RStudio environment
display(Image3, method="raster")
```

```
display(Image4, method="raster")
```

C. Gamma Correction

Gamma correction is the name of a nonlinear operation used to code and decode luminance or tristimulus values in video or still image systems, defined by the following power-law expression: begin{equation}nonumber V_{mathrm{out}} = AV_{mathrm{in}}^{gamma} end{equation} where A is a constant and the input and output values are non-negative real values; in the common case of $A = 1$, inputs and outputs are typically in the range 0-1. A gamma value $gamma < 1$ is sometimes called an encoding gamma (Wikipedia, Ref. 1)

```
Image5 <- Image.256 ^ 2
Image6 <- Image.256 ^ 0.7

# Let's display them in our RStudio environment
display(Image5, method="raster")
```

```
display(Image6, method="raster")
```

C. Cropping

Slicing array of pixels, simply mean cropping the image

```
display(Image.256[100:200, 50:100,], method="raster")
```

```
# Or, once again, in your Google Chrome browser
# display(Image.256[100:200, 50:100,])
```

D. Spatial Transformation

Spatial manipulation like rotate (rotate), flip (flip), and translate (translate) are also available in the package.

```
Imagetr <- translate(rotate(Image.256, 45), c(50, 0))
display(Imagetr, method="raster")
```

E. Color Management

Since the array of pixels has three axes in its dimension, in our case these are 5184 x 3456 x 3. The third axis is the slot for the three channels: Red, Green and Blue, or RGB. Hence, transforming the color.mode from Color to Grayscale, implies disjoining the three channels from single rendered frame (three channels for each pixel) to three separate array of pixels for red, green, and blue frames.

```
colorMode(Image.256) <- Grayscale
print(Image.256)
display(Image.256, method="raster")
```

```
# Switch back to color
colorMode(Image.256) <- Color
display(Image.256, method="raster")
```

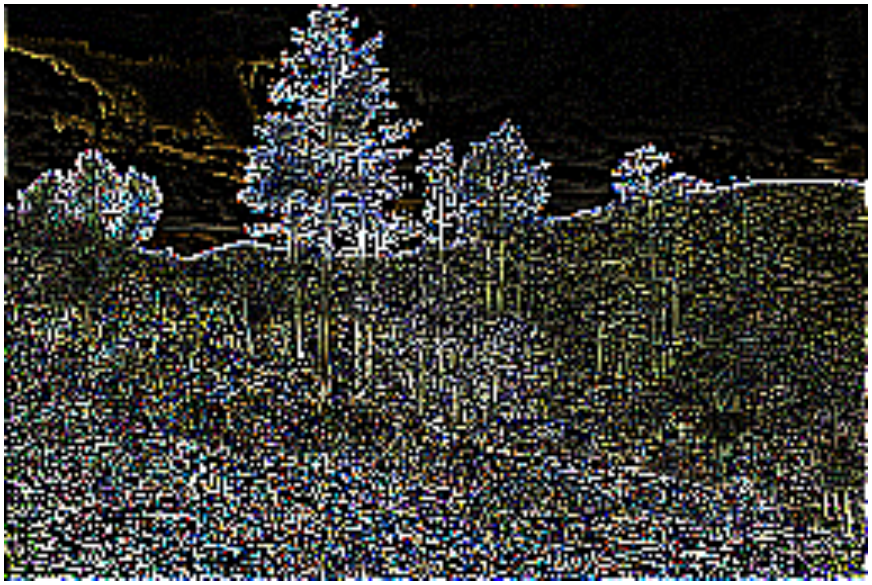
F. Filtering

Let's do smoothing/blurring using low-pass filter, and edge-detection using high-pass filter. In addition, we will also investigate median filter to remove noise.

```
fLow <- makeBrush(21, shape= 'disc', step=FALSE)^2
fLow <- fLow/sum(fLow)
Image.256.fLow <- filter2(Image.256, fLow)
display(Image.256.fLow, method="raster")
```

Or High Pass

```
fHigh <- matrix(1, nc = 3, nr = 3)
fHigh[2, 2] <- -8
Image.256.fHigh <- filter2(Image.256, fHigh)
display(Image.256.fHigh, method="raster")
```



Now go forth and do great things!