CSC-525: HPC SP2020 Final Report

Carrie Minette

2020-04-30

Contents


```
library(imageScrambleR)
```

1 Motivation

imageScrambleR is an image scrambling and similarity exercise written in R and C++. The primary purpose of this package was to provide an opportunity for the author to learn to write code that would utilize R's interface to C++ using the *Rcpp* package, and demonstrate the speed gains available through such interfacing. Such gains can be far more than trivial; one of R's most glaring downsides as a language is time inefficiency. Though R provides many optimizations within the language itself for experienced users in an attempt to address this, the fact remains that a trade off occurs, favoring ease of use over computational speed. When no further speed can be eked out through the use of R's native vectorization support and other tricks, the next option is to interface with an inherently faster lower level language such as C/C++.

2 Background

When initially proposed, *imageScrambleR* was a relatively ambitious project for the author, who had never before interfaced between any two languages, let alone between two languages with such notable paradigm differences. While R contains a C API that it utilizes under the hood, and extensions of this API such as the *Rcpp* package exist to provide an interface capable of compiling C++, R is, at its heart, a functional language based on the idea that anything that exists in R is an object, and anything that happens in R is a function. In contrast, C is an imperative language, with object-oriented support largely provided by C++. Thus, writing code

that is interoperable between R and C++ provided numerous design challenges, as the best methods for approaching any task can vary wildly between the two languages. These challenges were compounded by hardware complications encountered by the author, when some of the compiled C++ code from an R package utilized by another project the author was previously part of caused a conflict with a cloud-based file syncing service. This conflict eventually led a corrupted operating system pointer.The established methods to correct these problems instead compounded them, causing the issue to snowball, resulting in the author spending three days restoring the operating system multiple times, and losing more time as a software development workflow environment that had been built up over the course of several years needed to be rebuilt piece by piece.

3 Features

Having experienced first-hand the severity of the problems that can be caused by improper interactions between R, C++, and the surrounding development environment, the author was perhaps overly cautious, making every attempt to gain sufficient knowledge of the task at hand through extensive research before attempting to write code using *Rcpp*'s interface. The cumulative effect of these combined confounding factors led to software that falls short of its proposal due to time constraints. The current product lacks the similarity methods originally proposed, and the threaded image scrambling method is inoperable, crashing R on usage. Nonetheless, important strides were made. As it stands, the package features include: - Two convenience wrappers to facilitate converions between the format used by the image manipulation package *magick* and the raw hexadecimal matrices used by the package functions (*arr2magick* and *magick2arr*) - An image scrambler written in optimized R code (*scrambler_R*), - An image scrambler written in *Rcpp*-enabled C++ that emulates R design ideals (*scrambler_Cpp2*) - An image scrambler written in *Rcpp*-enabled C++ using object-oriented design ideals (*scrambler_Cpp*)

4 Results

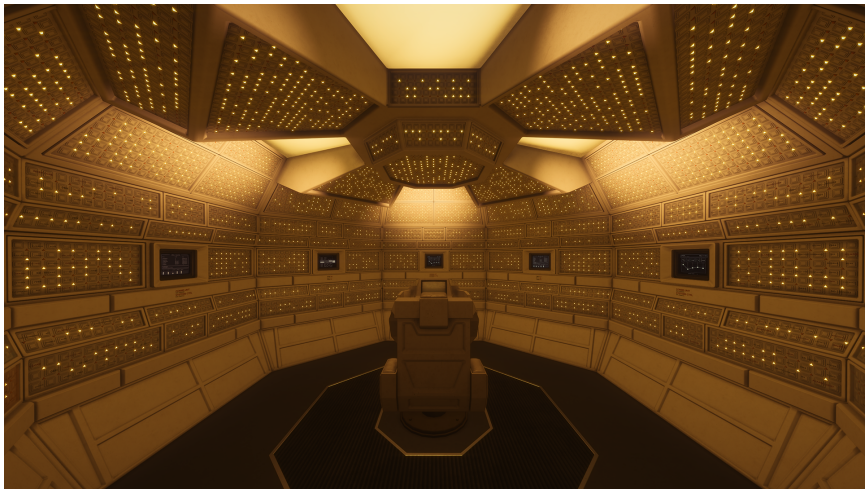
To demonstrate *imageScrambleR*'s current capabilities, the following code loads a variety of images using the *magick* package.

```
r_logo_path <- file.path(R.home("doc"), "html", "logo.jpg")
base_image_path <- "D:/feana/Pictures/"
queen75_path <- paste0(base_image_path, "queen75.jpg")
muthur_path <- paste0(base_image_path, "muthur.png")
sephiroth_path <- paste0(base_image_path, "sephiroth_uhd.jpg")

library(magick)
#> Linking to ImageMagick 6.9.9.14
#> Enabled features: cairo, freetype, fftw, ghostscript, lcms, pang
o, rsvg, webp
#> Disabled features: fontconfig, x11
r_logo <- image_read(r_logo_path)
print(r_logo)
#>   format width height colorspace matte filesize density
#> 1   JPEG   100     76       sRGB FALSE    15985 300x300
```

```
queen75 <- image_read(queen75_path)
print(queen75)
#>   format width height colorspace matte filesize density
#> 1   JPEG   550    363       sRGB FALSE    93663   96x96
```

```
muthur <- image_read(muthur_path)
print(muthur)
#>   format width height colorspace matte filesize density
#> 1    PNG  1920   1080       sRGB FALSE  2569483   72x72
```



```
sephiroth <- image_read(sephiroth_path)
print(sephiroth)
#>   format width height colorspace matte filesize density
#> 1   JPEG  3840   2160       sRGB FALSE  1067895   72x72
```

The *magick2arr* function is used to convert these images to grayscale, then to matrices with hexadecimal values for each pixel.

```
r_logo_arr <- magick2arr(r_logo)
print(dim(r_logo_arr))
#> [1] 100  76


queen75_arr <- magick2arr(queen75)
print(dim(queen75_arr))
#> [1] 550 363


muthur_arr <- magick2arr(muthur)
print(dim(muthur_arr))
#> [1] 1920 1080


sephiroth_arr <- magick2arr(sephiroth)
print(dim(sephiroth_arr))
#> [1] 3840 2160
```

We can then use any of the three image scrambler functions to scramble each image using an integer to indicate how many chunks we want each dimension split into.

```
r_logo_arr_scram <- scrambler_R(r_logo_arr, 3L)
queen75_arr_scram <- scrambler_Cpp2(queen75_arr, 5L)
muthur_arr_scram <- scrambler_Cpp(muthur_arr, 7L)
sephiroth_arr_scram <- scrambler_Cpp(sephiroth_arr, 9L)
```

Now we can use the *arr2magick* function to convert them back to *magick* image format, and view them.

```
r_logo_scram <- arr2magick(r_logo_arr_scram)
print(r_logo_scram)
#>   format width height colorspace matte filesize density
#> 1    PNG   100     76       Gray FALSE        0   72x72
```

```
queen75_scram <- arr2magick(queen75_arr_scram)
print(queen75_scram)
#>   format width height colorspace matte filesize density
#> 1    PNG   550    363       Gray FALSE        0   72x72
```

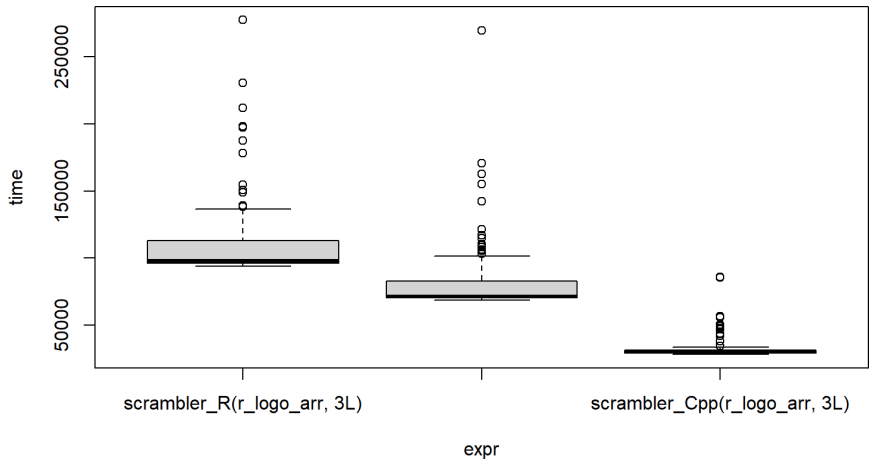
```
muthur_scram <- arr2magick(muthur_arr_scram)
print(muthur_scram)
#>   format width height colorspace matte filesize density
#> 1    PNG  1920   1080       Gray FALSE        0   72x72
```

```
sephiroth_scram <- arr2magick(sephiroth_arr_scram)
print(sephiroth_scram)
#>   format width height colorspace matte filesize density
#> 1    PNG  3840   2160       Gray FALSE        0   72x72
```

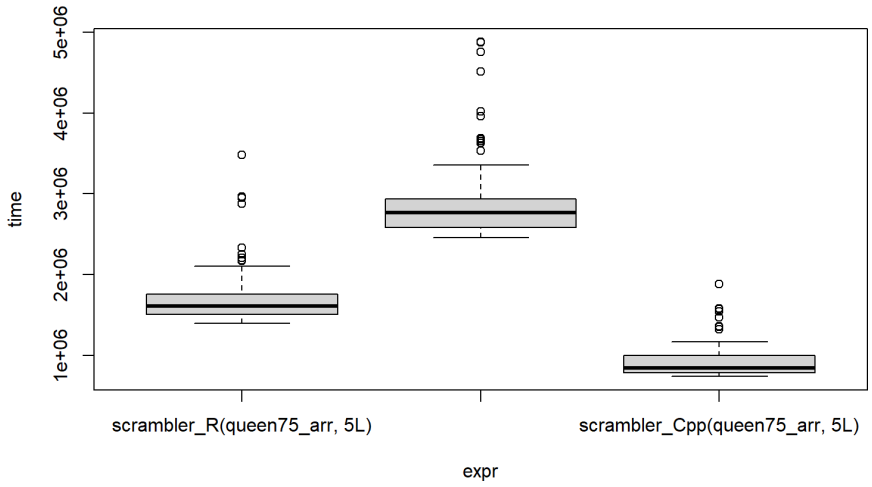
5 Benchmarks

The R package *microbenchmark* was used to run the three image scramblers 100 times each, using the same input parameters. Output time is in nanoseconds.

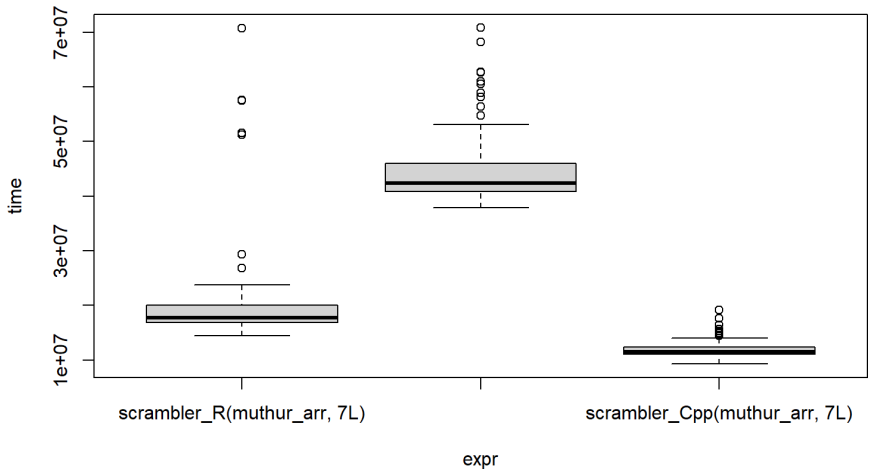
```
benchmarks_r_logo <- microbenchmark::microbenchmark(scrambler_R(r_l
ogo_arr, 3L),
                                                    scrambler_Cpp2(r_logo_
arr, 3L),
                                                    scrambler_Cpp(r_logo_a
rr, 3L))
print(benchmarks_r_logo)
#> Unit: microseconds
#>                               expr  min    lq    mean median     uq
max neval
#>     scrambler_R(r_logo_arr, 3L) 93.9 96.20 111.892  97.85 113.05
277.3   100
#>  scrambler_Cpp2(r_logo_arr, 3L) 68.5 70.40  82.677  71.45  82.85
269.5   100
#>   scrambler_Cpp(r_logo_arr, 3L) 28.3 29.25  33.827  30.10  31.35
85.8   100
plot(benchmarks_r_logo)
```




```
benchmarks_queen75 <- microbenchmark::microbenchmark(scrambler_R(qu
een75_arr, 5L),
                                                     scrambler_Cpp2(queen75
_arr, 5L),
                                                     scrambler_Cpp(queen75_
arr, 5L))
print(benchmarks_queen75)
#> Unit: microseconds
#>                                expr    min      lq     mean  median
uq    max
#>     scrambler_R(queen75_arr, 5L) 1398.3 1508.75 1705.487 1610.10
1754.45 3484.3
#>  scrambler_Cpp2(queen75_arr, 5L) 2458.7 2584.55 2900.045 2765.95
2936.80 4879.4
#>   scrambler_Cpp(queen75_arr, 5L)  738.5  786.85  928.103  842.70
996.15 1884.2
#>  neval
#>    100
#>    100
#>    100
plot(benchmarks_queen75)
```




```
benchmarks_muthur <- microbenchmark::microbenchmark(scrambler_R(mut
hur_arr, 7L),
                                                    scrambler_Cpp2(muthur_
arr, 7L),
                                                    scrambler_Cpp(muthur_a
rr, 7L))
print(benchmarks_muthur)
#> Unit: milliseconds
#>                               expr     min       lq     mean   medi
an       uq
#>     scrambler_R(muthur_arr, 7L) 14.4172 16.87285 20.39792 17.742
60 20.07700
#>  scrambler_Cpp2(muthur_arr, 7L) 37.9399 40.89390 44.82040 42.405
95 45.99095
#>   scrambler_Cpp(muthur_arr, 7L)  9.2865 11.03690 12.02306 11.550
55 12.37390
#>      max neval
#>  70.7184   100
#>  70.8185   100
#>  19.1177   100
plot(benchmarks_muthur)
```



```
benchmarks_sephiroth <- microbenchmark::microbenchmark(scrambler_R
(sephiroth_arr, 9L),
                                                       scrambler_Cpp2(sephiro
th_arr, 9L),
                                                       scrambler_Cpp(sephirot
h_arr, 9L))
print(benchmarks_sephiroth)
#> Unit: milliseconds
#>                                  expr      min        lq      mean
median
#>     scrambler_R(sephiroth_arr, 9L)  62.8083  64.77015  72.81005
67.31020
#>  scrambler_Cpp2(sephiroth_arr, 9L) 232.8872 238.06860 244.21409
 241.79160
#>   scrambler_Cpp(sephiroth_arr, 9L)  86.7876  89.06315  90.98079
90.45815
#>         uq      max neval
#>   71.58265 120.8485   100
#>  247.00245 289.2791   100
#>   91.92740  99.9329   100
plot(benchmarks_sephiroth)
```