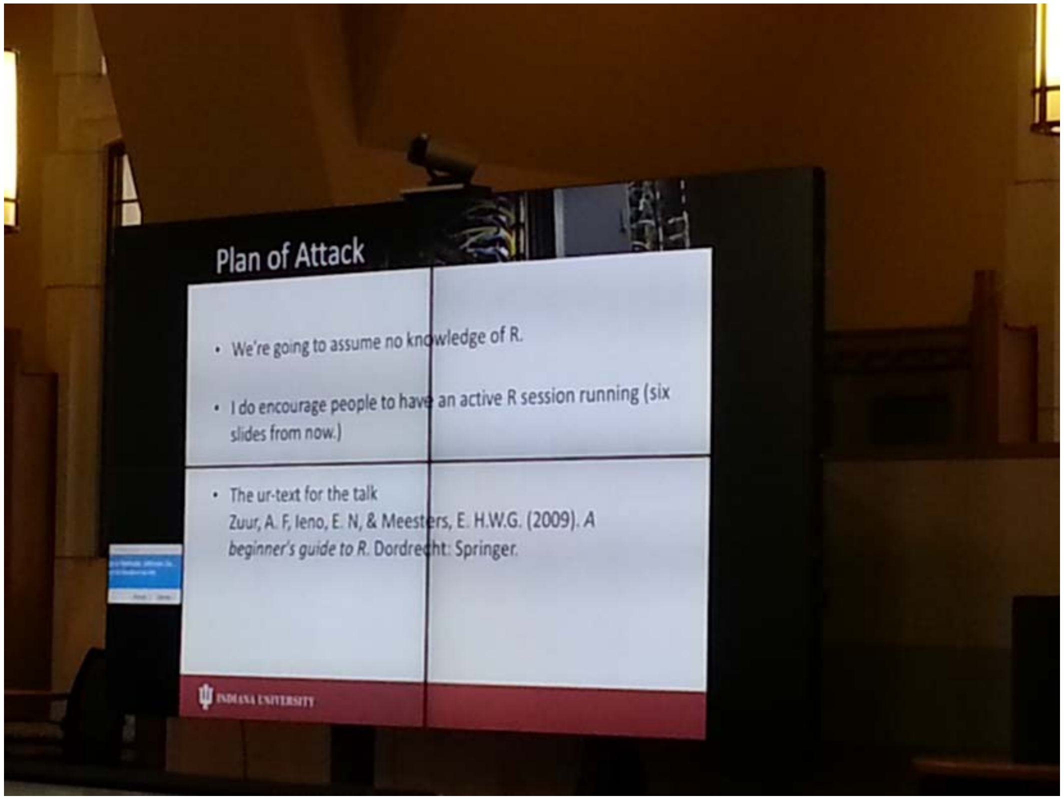
Writing Better R Code

WIM Oct 30, 2015

Hui Zhang
Research Analytics

“Introduction to R” by Jefferson, Oct 23

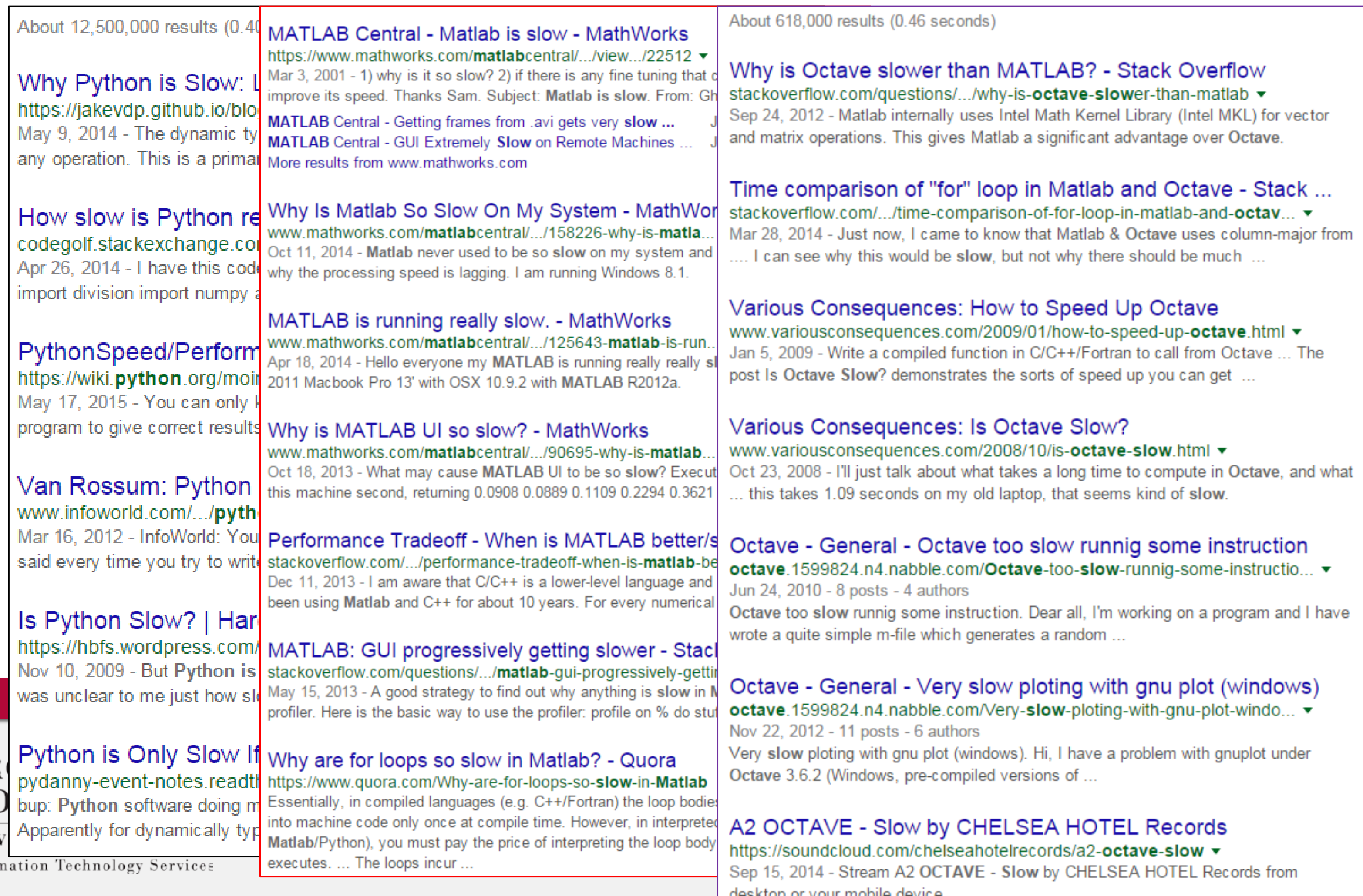


We all love R

- *Interactive data analysis*
- *Data mining/Machine Learning*
- *Plotting/Interactive 3D graphics*
- *Data-intensive Computing*

R is SLOW

- For the same reason any other interpret languages are



R is SLOW

- For the same reason any other interpret languages are slow
- R is optimized to make programmer efficient (instead of making machine efficient)
- Every single operation carries a lot of extra baggage

```r
#############################################################################
# Example 1.1
#############################################################################
system.time({
  I = 0
  while (I < 100000) {
    10
    I = I + 1
  }
})
```

Write Better R Codes

- Approaches for improving the performance of R codes
  - Some previous knowledge of R is recommended
  - Some familiarity with C/C++ is also recommended.
- Topics
  - Loops
  - Ply Functions
  - Vectorization
  - Loop, Plys, and Vectorization
  - Interfacing to Compiled Code

Loops

- Writing Better R Code
  - **Loops**
  - Ply Functions
  - Vectorization
  - Loop, Plys, and Vectorization
  - Interfacing to Compiled Code

Loops

- Writing Better R Code
  - **Loops**
    - **for**
    - **while**
    - No **goto's** or **do while's**
    - They are *really* slow
      - Why?
        - » for the same reason any interpreted language is slow
        - » every single operation carries a lot of extra baggage
        - » **particularly slow if objects grow inside Loops**

Loops

- Writing Better R Code
  - **Loops**
    - **Best Practices**
      - Mostly try to avoid
      - Evaluate practicality of rewrite (plys, vectorization, compiled code)
      - Always preallocate (avoid growing objects in loops):
        - » Vectors: numeric(n), integer(n), character(n)
        - » Lists: vector(mode="list", length=n)
        - » Dataframes: data.frame(col1=numeric(n), ...)
      - If you can't, try something other than an array/list.

Loops

```
square_loop_noinit <- function(n){
  x <- c()
  for (i in 1:n){
    x <- c(x, i^2)
  }

  x
}


square_loop_withinit <- function(n){
  x <- integer(n)
  for (i in 1:n){
    x[i] <- i^2
  }

  x
}
```

Loops

Rbenchmark

- *inspired by the Perl module [Benchmark]*
- *facilitate benchmarking of arbitrary R code*
- benchmark(..., replications=100, ..., relative = “elapsed”)

```
library(rbenchmark)
n <- 1000

benchmark(square_loop_noinit(n), square_loop_withinit(n))
#                       test replications elapsed relative
# 1    square_loop_noinit(n)          100   0.257    2.596
# 2 square_loop_withinit(n)          100   0.099    1.000
```

Ply Fucntions

- Writing Better R Code
  - Loops
  - **Ply Functions**
  - Vectorization
  - Loop, Plys, and Vectorization
  - Interfacing to Compiled Code

Ply Functions

- Writing Better R Code
  - Loops
  - **Ply Functions**
    - R has functions that apply other functions to data
    - In a nutshell: loop sugar
    - Typical *ply's
      - **apply()**: apply function over matrix "margin(s)"
      - **lapply()**: apply function over list/vector
      - **mapply()**: apply function over multiple lists/vectors
      - **sapply()**: same as lapply(), but (possibly) nicer output
      - Plus some other mostly irrelevant ones

Ply Functions

```r
x <- matrix(1:10, 2)

x
#      [,1] [,2] [,3] [,4] [,5]
# [1,]    1    3    5    7    9
# [2,]    2    4    6    8   10
```

Ply Functions

```
lapply(1:4, sqrt)
# [[1]]
# [1] 1
#
# [[2]]
# [1] 1.414214
#
# [[3]]
# [1] 1.732051
#
# [[4]]
# [1] 2

sapply(1:4, sqrt)
# [1] 1.000000 1.414214 1.732051 2.000000
```

Ply Functions

- Writing Better R Code
  - Loops
  - **Ply Functions**
    - Transforming Loops into Ply's

```
vec <- numeric(n)
for (i in 1:n){
  vec[i] <- my_function(i)
}
```

Becomes:

```
sapply(1:n, my_function)
```

Ply Functions

- Writing Better R Code
  - Loops
  - **Ply Functions**
    - Most Ply's are just shorthand/higher expression of loops
    - Generally not much faster (if at all), especially with the compiler
    - Thinking in terms of lapply() can be useful however …

Vectorization

- Writing Better R Code
  - Loops
  - Ply Functions
  - **Vectorization**
  - Loop, Plys, and Vectorization
  - Interfacing to Compiled Code

Vectorization

- Writing Better R Code
  - Loops
  - Ply Functions
  - **Vectorization**
    - In R *everything* is a vector. To quote Tim Smith in ***aRrgh: a newcomer's (angry) guide to R***
      - *"All naked numbers are double-width floating-point atomic vectors of length one. You're welcome."*
    - X + Y
    - X[, 1] <- 0
    - Rnorm(1000)

Vectorization

- Writing Better R Code
  - Loops
  - Ply Functions
  - **Vectorization**
    - also true in other high-level languages (Matlab, Python, ...)
    - Idea:
      - write vectorized code X[, 1] <- 0
      - use pre-existing compiled kernels to avoid interpreter overhead

Rnorm(1000)

    - Much faster than loops and plys

Vectorization

- Writing Better R Code
  - Loops
  - Ply Functions
  - **Vectorization**

```r
ply <- function(x) lapply(rep(1, 1000), rnorm)
vec <- function(x) rnorm(1000)

library(rbenchmark)
benchmark(ply(x), vec(x))
#       test replications elapsed relative
# 1 ply(x)          100   0.348   38.667
# 2 vec(x)          100   0.009    1.000
```

Vectorization

- Writing Better R Code
  - Loops
  - Ply Functions
  - **Vectorization**
    - Best Practices
      - Vectorize if at all possible
        - » Note that this consumes potentially a lot of memory

Ply Fucntions

- Writing Better R Code
  - Loops
  - Ply Functions
  - Vectorization
  - **Loop, Plys, and Vectorization**
  - Interfacing to Compiled Code

Putting It All Together

- Writing Better R Code
  - Loops
  - Ply Functions
  - Vectorization
  - **Loop, Plys, and Vectorization**
    - Loops are slow
    - apply() are just for loops
    - Ply functions are not vectorized
    - Vectorization is fastest, but often needs a lot of memory

Putting It All Together

- Writing Better R Code
  - Loops
  - Ply Functions
  - Vectorization
  - **Loop, Plys, and Vectorization**
    - Example: **let us compute the square of the number 1-100000**, using
      - **for** loop without preallocation
      - **for** loop with preallocation
      - **sapply()**
      - vectorization

Putting It All Together

```r
square_loop_noinit <- function(n){
  x <- c()
  for (i in 1:n){
    x <- c(x, i^2)
  }

  x
}


square_loop_withinit <- function(n){
  x <- integer(n)
  for (i in 1:n){
    x[i] <- i^2
  }

  x
}
```

Putting It All Together

```
square_sapply <- function(n) sapply(1:n, function(i) i^2)

square_vec <- function(n) (1:n)*(1:n)
```

```
library(rbenchmark)
n <- 100000

benchmark(square_loop_noinit(n), square_loop_withinit(n),
    square_sapply(n), square_vec(n))
#                      test replications elapsed relative
# 1   square_loop_noinit(n)          100  17.296 2470.857
# 2 square_loop_withinit(n)          100   0.933  133.286
# 3        square_sapply(n)          100   1.218  174.000
# 4           square_vec(n)          100   0.007    1.000
```

RESEARCH TECHNOLOGIES
INDIANA UNIVERSITY
University Information Technology Services

PERVASIVE TECHNOLOGY INSTITUTE
INDIANA UNIVERSITY

Rcpp

- Writing Better R Code
  - Loops
  - Ply Functions
  - Vectorization
  - Loop, Plys, and Vectorization
  - **Interfacing to Compiled Code**

Rcpp

- Writing Better R Code
  - Loops
  - Ply Functions
  - Vectorization
  - Loop, Plys, and Vectorization
  - **Interfacing to Compiled Code**
    - R is mostly a C program
    - R extensions are mostly R programs
    - Rcpp is a API for you to access/extend/modify R object at C++ level

Rcpp

- Writing Better R Code
  - Loops
  - Ply Functions
  - Vectorization
  - Loop, Plys, and Vectorization
  - **Interfacing to Compiled Code**
    - Rcpp is:
      - R interface to compiled code
      - Package ecosystem
      - Utilities to make writing C++ more convenient for R users
      - A tool which requires C++ knowledge to effectively utilize
      - GPL licensed (like R)

Rcpp

- Writing Better R Code
  - Loops
  - Ply Functions
  - Vectorization
  - Loop, Plys, and Vectorization
  - **Interfacing to Compiled Code**
    - Rcpp is **not**
      - Magic
      - Automatic R-to-C++ converter
      - A way around having to learn C++
      - A tool to make existing R functionality faster (unless you rewrite it)
      - As easy to use as R

Rcpp

- Writing Better R Code
  - Loops
  - Ply Functions
  - Vectorization
  - Loop, Plys, and Vectorization
  - **Interfacing to Compiled Code**
    - Rcpp's advantage
      - Compiled code is fast
      - Easy to install
      - Easy to use (comparatively)
      - Better documented than alternatives
      - Large, friendly, helpful community

Rcpp

```
code <- '
#include <Rcpp.h>

// [[Rcpp::export]]
int plustwo(int n)
{
  return n+2;
}
'
```

Rcpp

Consider a function defined as

$$f(n) \quad \text{such that} \quad \begin{cases} n & \text{when} \quad n < 2 \\ f(n-1) + f(n-2) & \text{when} \quad n \geq 2 \end{cases}$$

Rcpp

```r
f <- function(n) {
    if (n < 2) return(n)
    return(f(n-1) + f(n-2))
}
## Using it on first 11 arguments
sapply(0:10, f)

##  [1]  0  1  1  2  3  5  8 13 21 34 55
```

Rcpp

```
library(rbenchmark)
benchmark(f(10), f(15), f(20))[,1:4]

##     test replications elapsed relative
## 1 f(10)          100   0.034     1.00
## 2 f(15)          100   0.394    11.59
## 3 f(20)          100   4.384   128.94
```

```
int g(int n) {
    if (n < 2) return(n);
    return(g(n-1) + g(n-2));
}
```

Deployed as:

```
library(Rcpp)
cppFunction('int g(int n) { if (n < 2)
return(n); return(g(n-1) + g(n-2)); }')
## Using it on first 11 arguments
sapply(0:10, g)

## [1] 0 1 1 2 3 5 8 13 21 34 55
```

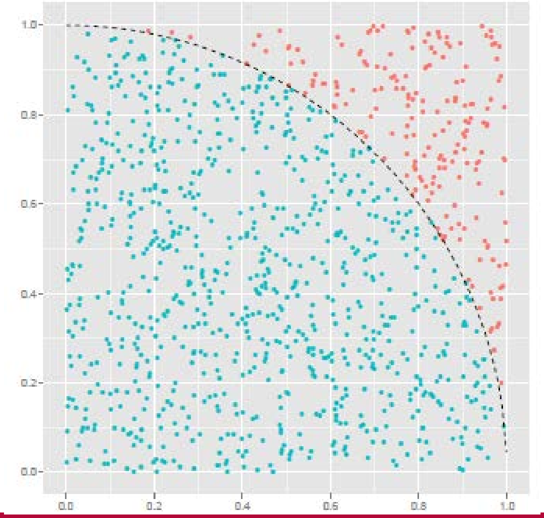
```r
library(rbenchmark)
benchmark(f(20), g(20))[,1:4]

##     test replications elapsed relative
## 1 f(20)          100   4.286    612.3
## 2 g(20)          100   0.007      1.0
```

Rcpp

- Example: Monte Carlo Simulation to Estimate π
  - Sample N uniform observation (x_i, y_i) in the unit square [0,1] X [0,1]. Then

$$\pi \approx 4\left(\frac{\#Inside\ Circle}{\#Total}\right)=4\left(\frac{\#Blue}{\#Blue+\#Red}\right)$$



Rcpp

- Example: Monte Carlo Simulation to Estimate π

R Code (loops)

```r
mcsim_r <- function(n)
{
  r <- 0L

  for (i in 1:n){
    u <- runif(1)
    v <- runif(1)

    if (u^2 + v^2 <= 1)
      r <- r + 1
  }

  return( 4*r/n )
}
```

Rcpp

- Example: Monte Carlo Simulation to Estimate π

R Code (vectorized)

```
mcsim_r_vec <- function(n)
{
  x <- matrix(runif(n * 2), ncol=2)
  r <- sum(rowSums(x^2) <= 1)

  return( 4*r/n )
}
```

Rcpp

- Example: Monte Carlo Simulation to Estimate π

Rcpp Code

```
code <- "
#include <Rcpp.h>

// [[Rcpp::export]]
double mcsim_rcpp(const int n)
{
  int i, r = 0;
  double u, v;

  for (i=0; i<n; i++){
    u = R::runif(0, 1);
    v = R::runif(0, 1);

    if (u*u + v*v <= 1)
      r++;
  }

  return (double) 4.*r/n;
}
"

library(Rcpp)
sourceCpp(code=code)
```

Rcpp

Benchmarking the Methods

```
library(rbenchmark)
library(compiler)

mcsim_r <- cmpfun(mcsim_r)
mcsim_r_vec <- cmpfun(mcsim_r_vec)
mcsim_rcpp <- cmpfun(mcsim_rcpp)

n <- 100000L

benchmark(R.loop = mcsim_r(n),
          R.vec = mcsim_r_vec(n),
          Rcpp = mcsim_rcpp(n),
          columns=c("test", "replications", "elapsed",
              "relative"))
```

```
    test replications elapsed relative
3   Rcpp          100   0.311    1.000
1 R.loop          100  55.125  177.251
2  R.vec          100   1.107    3.559
```

More Tricks???

- So far we only use one CPU core for R codes
- It is possible to parallelize the computation in LOOPs/PLYs

More Tricks???

- So far we only use one CPU core for R codes
- It is possible to parallelize the computation in LOOPs/PLYs
  - How many cores you have?

```
> library(parallel)
> cores <- detectCores()
> cores
[1] 16
```

More Tricks???

- So far we only use one CPU core for R codes
- It is possible to parallelize the computation in LOOPs/PLYs
  - How many cores you have?
  - Thinking of your codes in terms of PLYs can be useful

```
> library(parallel)
> cores <- detectCores()
> cores
[1] 16

> system.time(lapply(1:3000, rnorm))
   user  system elapsed
  0.560   0.016   0.576
```

More Tricks???

- So far we only use one CPU core for R codes
- It is possible to parallelize the computation in LOOPs/PLYs
  - How many cores you have?
  - Thinking of your codes in terms of PLYs can be useful
  - library(parallel)
    - let each core do the job independently for you
    - collect the results from each slave core

```
> library(parallel)
> cores <- detectCores()
> cores
[1] 16

> system.time(lapply(1:3000, rnorm))
   user  system elapsed
  0.560   0.016   0.576

> system.time(mclapply(1:3000, rnorm, mc.cores=cores))
   user  system elapsed
  0.390   0.100   0.233
```

More Tricks???

- So far we only use one CPU core for R codes
- It is possible to parallelize the computation in LOOPs/PLYs
  - How many cores you have?
  - Thinking of your codes in terms of PLYs can be useful
  - library(parallel)
    - let each core do the job independently for you
    - collect the results from each slave core
  - Note that there is overhead due to data shipping

```
> library(parallel)
> cores <- detectCores()
> cores
[1] 16

> system.time(lapply(1:3000, rnorm))
   user  system elapsed
  0.560   0.016   0.576

> system.time(mclapply(1:3000, rnorm, mc.cores=cores))
   user  system elapsed
  0.390   0.100   0.233
```

Summary

- Bad R often looks like good C/C++
- Vectorize your code as you much as you can
- Interfacing with compiled code helps
- Parallelization can take your code to extreme
- More reading:
  - “*The R Inferno*.” Patrick Burns
  - “*Rcpp: seamless R and C++ integration*. “ Dirk Eddelbuettel

Reading

- *"The R Inferno*." Patrick Burns
- *"Rcpp: seamless R and C++ integration*. " Dirk Eddelbuettel
- *"R and Data Mining: Examples and Case Studies."* Yanchang Zhao
- *"Data Visualization using R and Javascript."* Tom Barker
- *"Parallel R."* Ethan McCallum