`apply()` Yourself

Purpose and Implementation of `apply()` in R

Todd Wilson

Statistical Learning Group
Department of Statistics
North Carolina State University

November 28, 2016

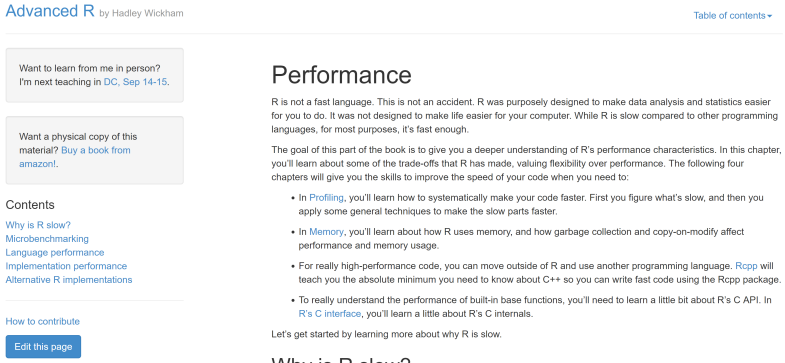
Presentation Outline

- Motivation for presentation
- "Why is R slow?"
- `for()` loops in R
- Introduction to `apply()` family of functions
- Implementation of `apply()` in R

Motivation

- 9/20: Dr. Post used `apply()` in his regression tree code
- 10/11: Josh Day discussed why Julia is fast
- $\sim$10/26: Todd struggled with `apply()` during spatial HW
- Seconds later: Todd decided to give the `apply()` talk

"Why is R slow?"



Advanced R by Hadley Wickham

Table of contents

Want to learn from me in person? I'm next teaching in DC, Sep 14-15.

Want a physical copy of this material? Buy a book from amazon!.

Contents

Why is R slow?
Microbenchmarking
Language performance
Implementation performance
Alternative R implementations

How to contribute

Edit this page

Performance

R is not a fast language. This is not an accident. R was purposely designed to make data analysis and statistics easier for you to do. It was not designed to make life easier for your computer. While R is slow compared to other programming languages, for most purposes, it's fast enough.

The goal of this part of the book is to give you a deeper understanding of R's performance characteristics. In this chapter, you'll learn about some of the trade-offs that R has made, valuing flexibility over performance. The following four chapters will give you the skills to improve the speed of your code when you need to:

- In Profiling, you'll learn how to systematically make your code faster. First you figure what's slow, and then you apply some general techniques to make the slow parts faster.
- In Memory, you'll learn about how R uses memory, and how garbage collection and copy-on-modify affect performance and memory usage.
- For really high-performance code, you can move outside of R and use another programming language. Rcpp will teach you the absolute minimum you need to know about C++ so you can write fast code using the Rcpp package.
- To really understand the performance of built-in base functions, you'll need to learn a little bit about R's C API. In R's C interface, you'll learn a little about R's C internals.

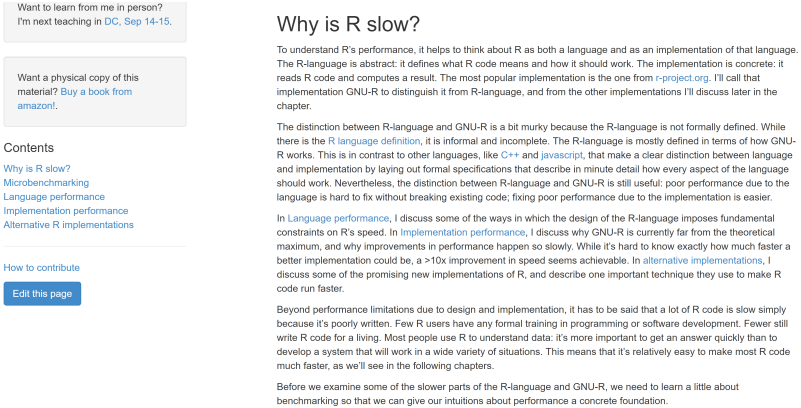
Let's get started by learning more about why R is slow.

Why is R slow?

To understand R's performance, it helps to think about R as both a language and as an implementation of that language. The R-language is abstract: it defines what R code means and how it should work. The implementation is concrete: it reads R code and computes a result. The most popular implementation is the one from r-project.org. I'll call that implementation GNU-R to distinguish it from R-language, and from the other implementations I'll discuss later in the

Image source: http://adv-r.had.co.nz/Performance.html

"Why is R slow?"



Want to learn from me in person?
I'm next teaching in DC, Sep 14-15.

Want a physical copy of this material? Buy a book from amazon!.

Contents

How to contribute

Edit this page

Why is R slow?

To understand R's performance, it helps to think about R as both a language and as an implementation of that language. The R-language is abstract: it defines what R code means and how it should work. The implementation is concrete: it reads R code and computes a result. The most popular implementation is the one from r-project.org. I'll call that implementation GNU-R to distinguish it from R-language, and from the other implementations I'll discuss later in the chapter.

The distinction between R-language and GNU-R is a bit murky because the R-language is not formally defined. While there is the R language definition, it is informal and incomplete. The R-language is mostly defined in terms of how GNU-R works. This is in contrast to other languages, like C++ and javascript, that make a clear distinction between language and implementation by laying out formal specifications that describe in minute detail how every aspect of the language should work. Nevertheless, the distinction between R-language and GNU-R is still useful: poor performance due to the language is hard to fix without breaking existing code; fixing poor performance due to the implementation is easier.

In Language performance, I discuss some of the ways in which the design of the R-language imposes fundamental constraints on R's speed. In Implementation performance, I discuss why GNU-R is currently far from the theoretical maximum, and why improvements in performance happen so slowly. While it's hard to know exactly how much faster a better implementation could be, a >10x improvement in speed seems achievable. In alternative implementations, I discuss some of the promising new implementations of R, and describe one important technique they use to make R code run faster.

Beyond performance limitations due to design and implementation, it has to be said that a lot of R code is slow simply because it's poorly written. Few R users have any formal training in programming or software development. Fewer still write R code for a living. Most people use R to understand data: it's more important to get an answer quickly than to develop a system that will work in a wide variety of situations. This means that it's relatively easy to make most R code much faster, as we'll see in the following chapters.

Before we examine some of the slower parts of the R-language and GNU-R, we need to learn a little about benchmarking so that we can give our intuitions about performance a concrete foundation.

Image source: http://adv-r.had.co.nz/Performance.html

A Simple Answer

- "R is not a fast language. This is not an accident."
- R makes statistics simpler for the user, not the computer
- "For most purposes, its fast enough."
- When you type `a <- 3.00` in R, what don't you type?

A Simple Answer

- "R is not a fast language. This is not an accident."
- R makes statistics simpler for the user, not the computer
- "For most purposes, its fast enough."
- When you type `a <- 3.00` in R, what don't you type?
  - 3.00 is a floating-point number

A Simple Answer

- "R is not a fast language. This is not an accident."
- R makes statistics simpler for the user, not the computer
- "For most purposes, its fast enough."
- When you type `a <- 3.00` in R, what don't you type?
  - 3.00 is a floating-point number
  - `a` will store data of type numeric

A Simple Answer

- "R is not a fast language. This is not an accident."
- R makes statistics simpler for the user, not the computer
- "For most purposes, its fast enough."
- When you type `a <- 3.00` in R, what don't you type?
  - 3.00 is a floating-point number
  - `a` will store data of type numeric
  - 3.00 needs to be stored in memory, with `a` as its pointer

A Simple Answer

- "R is not a fast language. This is not an accident."
- R makes statistics simpler for the user, not the computer
- "For most purposes, its fast enough."
- When you type `a <- 3.00` in R, what don't you type?
  - 3.00 is a floating-point number
  - `a` will store data of type numeric
  - 3.00 needs to be stored in memory, with `a` as its pointer
  - $3.00 =_2$ 00110011 00101110 00110000 00110000

(Start to) a More Technical Answer

- R is a high-level *interpreted* computer language
  - C, FORTRAN, many other faster languages are *compiled*
- Interpreted languages answer each "question" as they come, one at a time, without expecting you to answer them
- Compiled languages begin figuring out answers immediately, and you can help by answering some of the questions
- Josh taught us that Julia is a compiled language
  - But, "just-in-time" execution gives it the feel of an interpreted language

`for()` Loops Sake!

- Simple and intuitive, but clunky in an interpreted language
  - R must ask itself all questions, every time. (Every. Single. Time.)
  - Mistakes are easy!
  - Big data?
  - What about that `i` floating around in your workspace?
- In spite of this, there are times we must `for()` ge ahead
  - Each iteration depends on the previous one
  - Certain functions won't accept vector arguments
- Work-arounds: `ifelse()`, `apply()` family of functions

`for()` Loops Sake!

- Doesn't something have to loop?
- Well, yes, but not in an interpreted language...
- The `apply()` family hands off this task from R to C or FORTRAN, where it can be done more quickly!

Functional Programming

- "R is, at its heart, a functional programming language."
- R provides a giant toolbox to create/manipulate functions
- Instead of taking vectors as arguments, why not take functions as arguments?
- Such functions are called *functionals*
- Often, R programmers first encounter functionals with the `apply()` family

Let's `apply()` This!

- Situation: We wish to perform operations on structured data (i.e., a matrix).
- Structure: `apply(X, MARGIN, FUN, ...)`
  - X - an array, such as a vector or a matrix.
  - MARGIN (if a matrix) - a vector giving the subscripts which the function will be applied over. 1 = rows, 2 = columns, c(1, 2) = rows and columns, "dimnames" (if applicable).
  - FUN - the function to be applied.
- Strategy: Understand data structure to know which apply function to use.

??apply

Search Results 

Help pages:

base::apply	Apply Functions Over Array Margins
base::.subset	Internal Objects in Package 'base'
base::by	Apply a Function to a Data Frame Split by Factors
base::eapply	Apply a Function Over Values in an Environment
base::lapply	Apply a Function over a List or Vector
base::mapply	Apply a Function to Multiple List or Vector Arguments
base::rapply	Recursively Apply a Function to a List
base::tapply	Apply a Function Over a Ragged Array

(Move to RMarkdown file.)

Conclusion

- R is slow, but it's supposed to be, and we love it anyway
- `for()` loops are computationally cumbersome for R
- Because R is a functional programming language, function families like `apply()` can work around these issues
- There are many types of `apply()` functions, and which one to use depends on data types and desired output
- Further study: vectorizing code, `subset()`, `by()`, `replicate()`, `aggregate()`, `dplyr` package

References

- Rokicki, Slawa. "For Loops (and How to Avoid Them)." *R-bloggers*. 14 Jan. 2013. Web. 27 Nov. 2016.
- Ross, Noam. "Vectorization in R: Why?" *Real-time Research in Ecology, Economics, and Sustainability*. 16 Apr. 2014. Web. 27 Nov. 2016.
- Saunders, Neil. "A Brief Introduction to apply in R." *What You're Doing Is Rather Desperate*. 27 Feb. 2014. Web. 27 Nov. 2016.
- Wickham, Hadley. *Advanced R*. Boca Raton, FL: CRC, 2015. Print.