**Software engineering principles to improve quality and performance of R software**

Seth Russell1, Tellen D. Bennett1,2, Debashis Ghosh1,3

1 University of Colorado Data Science to Patient Value (D2V), Anschutz Medical Campus, Aurora, CO, United States of America

2 Pediatric Critical Care, University of Colorado School of Medicine, Aurora, CO, United States of America

3 Biostatistics and Informatics, Colorado School of Public Health, Aurora, CO, United States of America

Corresponding Author:

Seth Russell

CU Data Science to Patient Value

13199 E. Montview Blvd, Suite 210-15

Aurora, CO 80045 USA

Email address: seth.russell@ucdenver.edu

**ABSTRACT**

Today’s computational researchers are expected to be highly proficient in using software to solve a wide range of problems ranging from processing large datasets to developing personalized treatment strategies from a growing range of treatment options. While these researchers are well versed in their own field, formal training in software engineering principles and appropriate mentorship is often lacking. Two major themes that are not covered in most university coursework nor current literature are software testing and software optimization. Through a survey of all currently available CRAN (The Comprehensive R Archive Network) packages we show that reproducible and replicable software tests are not available, and that many packages do not appear to employ software performance and optimization tools and techniques. Through use of examples from an existing R package, we demonstrate powerful testing and optimization techniques that can improve the quality of any researcher’s software.

**INTRODUCTION**

**Abstract Guidance – remove this box before submitting!**

* No more than approx. 500 words (or 3,000 characters).
* Self-contained and concisely describe the reason for the work, methodology, results, and conclusions. Uncommon abbreviations should be spelled out at first use. Do not include footnotes or references.
* Headings in structured abstracts should be bold and followed by a period. Each heading should begin a new paragraph. For example:

**Background.** The background section text goes here. Next line for new section.

**Methods.** Methods section here, then new line.

**Results.** Results section here, then new line.

**Discussion.** Discussion section here.

Writing scientific software has progressed from the work of early pioneers to a range of computer professionals, computational researchers, and autodidacts. The educational discipline of computer science, codified through recommendations from the Association for Computing Machinery (ACM) (Atchison et al., 1968), has grown in breadth and depth over many years. There are now many programs and disciplines through which one can learn how carry out scientific research with the assistance of computers. Software engineering, an offshoot of computer science, is the discipline that “seeks to develop and use systematic models and reliable techniques to produce high-quality software. These concerns extend from theory and principles to the development practices that are most visible to those outside the discipline.” (The Joint Task Force on Computing Curricula, 2015)

In a similar manner to the advances in computer science body of knowledge, computational researchers, statisticians, and similar professionals need to advance their skills by adopting principles of software engineering. Wilson et al in their 2014 paper covered key areas where scientists can benefit from software engineering best practices (Wilson et al., 2014). They provide a high-level description of 8 important principles of software engineering that should “reduce the number of errors in scientific software, make it easier to reuse, and save the authors of the software time and effort that can used for focusing on the underlying scientific questions.” While their principles are still relevant and important today, there has not been enough progress in this endeavor, especially with respect to software testing and software optimization principles (Wilson, 2016; Nolan & Padilla-Parra, 2017).

The ACM/IEEE recommendations for an undergraduate degree in software engineering describe a range of course work and learning objectives. Their guidelines call out 10 specific knowledge areas that should be part of or guide all software engineering coursework. The major areas are: computer science fundamentals, math and engineering fundamentals, professional interactions/communication, software modeling, requirement gathering, software design, verification, project processes, quality, and security (The Joint Task Force on Computing Curricula, 2015). These major themes are not covered extensively outside software engineering and include such generally applicable items such as software verification, validation, testing, and computer science fundamentals (e.g. software optimization, modeling, and requirement gathering).

In addition to the need for further training, understanding the software lifecycle is necessary: the process of software development from ideation to delivery of code. The largest component of a software’s lifecycle is maintenance Software maintenance costs are large and increasing (Glass, 2001; Dehaghani & Hajrahimi, 2013; Koskinen, Jussi, 2015); some put maintenance at 90% of total software cost. The chief factor in cost of maintenance with respect to research and statistical software is time of the people creating and using the software. From the recent trend on making research results reproducible and replicable, some recommend making code openly available to any who might wish to repeat or further analyze results (Leek & Peng, 2015). With the development of any software artifact, an important consideration for implementation should be maintenance. As research scientists tend to think of their software products as unique tools that will not be used regularly or for a long period, they often fail to consider long term maintenance issues during the development phase. While a rigorous and formal software engineering approach is not well suited to the standard lifecycle of research software (Wilson, 2016), there are many techniques that can help to reduce cost of maintenance and speed development. While best practices such as the use of version control software, open access to data, software, and results are becoming more wide spread, other best practices such as testing and optimization need further attention.

In this paper, a brief survey of currently available R packages from CRAN will be used to show the continued need for software testing and optimization. Source code for this analysis is freely available at <https://github.com/magic-lantern/SoftwareEngineeringPrinciples>. Additionally, the R package “pccc: Pediatric Complex Chronic Conditions” (DeWitt et al., 2017) (PCCC), available via CRAN and at <https://github.com/CUD2V/pccc>, is used for code examples in this article. PCCC is a combined R and C++ implementation of the Pediatric Complex Chronic Conditions software released as part of a series of research papers (Feudtner, Christakis & Connell, 2000; Feudtner et al., 2014). The PCCC package takes as input a data set containing International Statistical Classification of Diseases and Related Health Problems (ICD) Ninth revision or Tenth revision diagnosis and procedure codes and outputs which if any complex chronic conditions a patient has.

**SOFTWARE TESTING**

Whenever software is written as part of a research project, careful consideration should be given on how to verify that the software performs the desired functionality and produces the desired output. As with bench science, software can often have un-expected and un-intended results due to minor or even major problems during the implementation process. In the software engineering field, software testing is a major component of any software development lifecycle and should also be a key component of research software. As will be shown later, even among R software packages intended to be shared with and used by others, the majority of R packages (67% to 74% depending on metric, see next section) do not have tests that are made available with the package. From this author’s experience, it is estimated that the amount of research software that has been formally tested is much lower.

Various methodologies and strategies exist for software testing and validation as well as how best to integrate software with a software development lifecycle. Some common testing strategies are no strategy, ad hoc testing (Agruss & Johnson, 2000), test driven development (TDD) (Kent Beck & Erich Gamma, 1998). There are also common project methodologies where testing fits into the project lifecycle; two common examples are waterfall, where testing is a major phase that occurs at a specific point in time, and agile, where there are many small iterations with testing. While a full discussion of various methods and strategies is beyond the scope of this article, following the “Analysis of testing of R Packages on CRAN” section, three key concepts presented are: when to start testing, what to test, and how to test.

**ANALYSIS OF TESTING OF R PACKAGES ON CRAN**

In order to get an estimate of the level of testing common among R software, an analysis of R packages available through CRAN has been performed. Although Nolan (Nolan & Padilla-Parra, 2017) has performed a similar analysis in the past, due to the rapid change in the CRAN as a whole, a reevaluation is necessary. At the time of Nolan’s work, CRAN contained 10084 packages, now it contains 13094. Furthermore, the analysis by Nolan had a few shortcomings that we have addressed in this analysis: there are additional testing frameworks for which we wanted to analyze their usage; not all testing frameworks and R packages store their test code in a directory named “tests”; only packages modified in the past 2 years was reported - there are many commonly used R packages that have not been updated in the last 2.

Although we address some shortcomings in analyzing R code for use of testing best practices, our choice of domain for analysis does have some limitations. Not all research software is written in R; for those that do use R, not all software development results in a package published on CRAN. While other software languages have tools for testing, additional research would be needed to evaluate level of testing in those languages to see how it compares to this analysis. Although care has been taken to identify standard testing use cases and practices for R, testing can be performed in-line through use of core functions such as stop() or stopifnot(). Also, developers may have their own test cases they run while developing their software, but did not include them in the package made available on CRAN. Unit tests can be considered as executable documentation, a key method of conveying how to use software correctly (jpreese). Published research that involves software is not as easy to access and evaluate for use of testing code as CRAN packages are. While some journals have standardized means for storing and sharing code, many leave the storing and sharing of code up to the individual author, creating an environment where code analysis would require significant manual effort.

To analyze use of software testing techniques, we evaluated all CRAN packages on two different metrics:

Metric 1: Grep for non-empty testing directories using the pattern "[Tt]est[^/]\*/.+". All commonly used R testing packages (those identified for metric 2) recommend placing tests in a directory by themselves, which we look for.

Metric 2: Check for stated dependencies on one of the following testing packages: RUnit (Burger, Juenemann & Koenig, 2015), svUnit (Grosjean, 2014), testit (Xie, 2018), testthat (Wickham, 2011), unitizer (Gaslam, 2017), or unittest (Lentin & Hennessey, 2017). It is considered best practice to list dependencies on a testing framework even though standard usage of a package may not require it.

For the testing analysis, we used 2008 as the cutoff year for visualizations due to the low number of packages updated prior to 2008.

As shown in Figure 1, the evaluation for the presence of a non-empty testing directory shows that there is an increasing trend in testing R packages, with 44% of packages updated in 2018 having some tests. Table S1 contains the data used to generate Figure 1.

As shown in Figure 2, reliance upon testing frameworks is increasing overtime both in count and as a percent of all packages. There 16 packages that list dependencies to more than one testing framework (9 with dependencies on both RUnit and testthat, 7 with dependencies on both testit and testthat), so the total number of packages shown in the histogram includes 16 that are double counted. Table S2 contains the data used to generate Figure 2.

As the numbers from Metric 1 do not match the numbers of Metric 2, some additional exploration is necessary. There are 866 more packages identified from Metric 1 vs Metric 2. There are 1080 packages that do not list a dependency to a testing framework, but have a testing directory – see the package xlsx (Dragulescu & Arendt, 2018) as an example. Some also actually use a testing framework, but do not list it as a dependency, see the package redcapAPI (Nutter & Lane, 2018) as an example. There are also 214 packages that list a testing framework as a dependency, but do not contain a directory with tests.

**WHEN TO TEST**

One of the popular movements in recent years has been to develop tests first then work to implement code to meet desired functionality, a strategy called test driven development. While the test driven development strategy has done much to improve the focus of the software engineering world on testing, some have found that it does not meet all development styles (David Heinemeier Hansson, 2014; Sommerville, 2016). A better approach that matches the flexible and changing nature of research software is to create tests after a requirement or feature has been implemented. As developing comprehensive tests of software functionality can be a large burden to accrue at a single point in time, a more pragmatic approach is to alternate between developing new functionality and designing tests to validate new functionality. Similar to the agile software development strategy, a build/test cycle can allow for quick cycles of validated functionality that help to provide input into additional phases of the software lifecycle.  
  
**WHAT TO TEST**

In an ideal world, any software developed would be accompanied by 100% test coverage validating all aspects of functionality and interaction with other software. However, due to pressures of research, having time to build a perfect test suite is not realistic. A parsimonious application of the Pareto principle will go a long way towards improving overall software quality without adding to the testing burden. Large companies such as Microsoft have applied traditional scientific methods to the study of bugs and found that the Pareto principle matches reality: 20% of bugs cause 80% of problems; additionally a Zipfian distribution may apply as well: 1% of bugs cause 50% of all problems (Rooney, 2002).

To apply the Pareto principle to testing, spend some time in a thought experiment to determine answers to questions such as: What is the most important feature(s) of this software? If this software breaks, what is the most likely bad outcome? For computationally intensive components - how long should this take to run?

Once answers to these questions are known, the developer(s) should spend time designing tests to validate key features, avoiding major negatives, and ensuring software performs adequately. Part of the test design process should include how to “test” more than just the code. Some specific aspects of non-code tests include validation of approach and implementation choices with a mentor or colleague.

**HOW TO TEST**

Most programming languages have a multitude of testing tools and frameworks available for assisting developers with the process of testing software. Due to the recurring patterns common across programming languages most languages have an SUnit (“SUnit,” 2017)(“SUnit,” 2017) derived testing tool, commonly referred to as an “xUnit” (“xUnit,” 2017) testing framework that focuses on validating individual units of code along with necessary input and output meet desired requirements. Based on software language used, unit tests may be at the class or function/procedure level. Some common xUnit style packages in R are RUnit and testthat. Unit tests should be automated and run regularly to ensure errors are caught and addressed quickly. For R, it is easy to integrate unit tests into the package build process, but other approaches such as via post-commit hook in a version control system are common.

In addition to unit tests, users should perform acceptance tests, or high-level functionality tests that validate the software meets requirements. Due to the high-level nature and subjective focus of acceptance tests, they are often manually performed. Careful documentation of how a user will actually use software, referred to as user stories, are translated into step by step tests that a human follows to validate the software works as expected. A few examples of acceptance testing tools are: Selenium (*Selenium*), Microfocus Unified Functional Testing (formely known as HP’s QuickTest Professional) (*Unified Functional Testing*), and Ranorex (*Ranorex*). As research focused software often does not have a GUI, an alternative to manual testing processes is to create a full step by step example via RMarkdown (Xie, 2015) notebook demonstrating use of the software followed by either manually or automatic validation that the expected end result is correct.

In addition to the tool-based approaches already mentioned, other harder to test items such as algorithms and solution approach should be scrutinized as well. While automated tests can validate mathematical operations or other logic steps are correct, they cannot verify that the approach or assumptions implied through software operations are correct. This level of testing can be done through code review and design review sessions with others who have knowledge of the domain or a related domain.

**TESTING ANTI-PATTERNS**

While the above guidance should help researchers know the basics of testing, it does not cover in detail what not to do. An excellent collection of testing anti-patterns can be found at (*Unit testing Anti-patterns catalogue*; Jarkko Moilanen; James Carr, 2015). Some key problems that novices experience when learning how to test software are:

* Interdependent tests - Interdependent tests can cause multiple test failures. When a failure in an early test case breaks a later test, it can cause difficulty in resolution and remediation.
* Testing application performance – While testing execution timing or software performance is a good idea and is covered more in the “Software Optimization” section, creating an automated test to perform this is difficult and does not carry over well from one machine to another.
* Slow running tests – As much as possible, tests should be automated but still run quickly. If the testing process takes too long consider refactoring tests or evaluating the performance of the software being tested.
* Only test correct input - A common problem in testing is to only validate expected inputs and desired behavior. Make sure tests cover invalid input, exceptions, and similar items.

**APPLICATION OF SOFTWARE TESTING TO R PACKAGE DEVELOPMENT**

As a brief example of how to apply the aforementioned testing principles, here is some information on key testing steps followed with PCCC. For the PCCC package there is a large set of ICD codes and code set patterns that are used to determine if an input record meets any complex chronic condition criteria. To validate the correct functioning of the software, the first priority was to validate the ICD code groupings were correct and were mutually exclusive (as appropriate). As PCCC is a re-implementation of existing SAS and Stata code, we needed to validate that the codes from the previously developed and published software applications were identical and were performing as expected. Through a combination of manual review and automated comparison codes were checked to see if duplicates and overlaps existed.

Here is a brief snippet of some of the code used to automatically find duplicates and codes that were already included as part of another code:

icds <- input.file("../pccc\_validation/icd10\_codes\_r.txt")  
  
unlist(lapply(icds, function(i) {  
 tmp <- icds[icds != i]  
 output <- tmp[grepl(paste0("^", i, ".\*"), tmp)]  
 # add the matched element into the output  
 if(length(output) != 0)  
 output <- c(i, output)  
 output  
}))

While the specific codes were being compared, validated, and agreed upon, unit tests were written to validate the functionality of the new package. The first tests written were those that were manually developed and manually run as development progressed. Key test cases of this form are ideal candidates for inclusion in automated testing. After these initial tests, further thought went into edge cases to check how the software might behave if the input data was incorrect or if parameters were not specified correctly.

Another common pattern is to create a test case for discovered bugs - this ensures that a re-occurrence, known as a “regression” to software engineers, of this error does not happen again. Here is an example that automated a test discovered due to the flexible nature of the sapply function returning different data types based on input:

# Due to previous use of sapply in ccc.R, this would fail - fixed now  
 test\_that("1 patient with multiple rows of no diagnosis data - should have all CCCs as FALSE", {  
 expect\_true(all(ccc(dplyr::data\_frame(id = 'a', dx1 = NA, dx2 = NA),  
 dx\_cols = dplyr::starts\_with("dx"),  
 icdv = code) == 0))  
 })

**SOFTWARE OPTIMIZATION**

Getting software to run in a reasonable amount of time is always a key consideration when working with large datasets. A mathematical understanding of software algorithms is usually a key component of computer science and software engineering curricula, but not widely covered in other disciplines. Additionally, while software engineering texts and curricula highlight the importance of testing for non-functional requirements such as performance (Sommerville, 2015), they often fail to provide details on how best to evaluate software performance or how to plan for performance during the various phases of software lifecycle.

The survey of R packages that follows indicates that approximately 75% of packages do not use optimization related packages nor compiled code to improve performance. Following the analysis of R packages, this section will provide a starting point for additional study, research, and experimentation. The key aspects of software optimization discussed in this are: identify a performance target, understanding and applying Big O notation, and the use code profiling and benchmarking tools.

**ANALYSIS OF OPTIMIZATION OF R PACKAGES ON CRAN**

In order to get an estimate of the level of software optimization common among R software, an analysis of R packages available through CRAN has been performed. To analyze the use of software optimization tools and techniques we evaluated all CRAN packages on two different metrics:

Metric 1: Grep for non-empty src directories. By convention, packages using compiled code (C, C++, Fortran) place those files in a '/src' directory.

Metric 2: Check for stated dependencies on packages that can optimize, scale performance, or evaluate performance of a package. Packages included in analysis are: DSL (Feinerer, Theussl & Buchta, 2015), Rcpp (Eddelbuettel & Balamuta, 2017), RcppParallel (Allaire et al., 2018), Rmpi (Yu, 2002), SparkR (“SparkR (R on Spark) - Spark 2.3.2 Documentation”), batchtools (Bischl et al., 2015), doMC (Calaway, Analytics & Weston, 2017), doMPI (Weston, 2017), doParallel (Calaway et al., 2018), doSNOW (Calaway, Corporation & Weston, 2017), foreach (Calaway, Microsoft & Weston, 2017), future (Bengtsson, 2018), future.apply (Bengtsson & R Core Team, 2018), microbenchmark (Mersmann, 2018), parallel (R Core Team, 2018), parallelDist (Eckert, 2018), parallelMap (Bischl & Lang, 2015), partools (Matloff, 2016), profr (Wickham, 2014a), profvis (Chang & Luraschi, 2018), rbenchmark (Kusnierczyk, Eddelbuettel & Hasselman, 2012), snow (Tierney et al., 2018), sparklyr (Luraschi et al., 2018), tictoc (Izrailev, 2014).

For the optimization analysis, we used 2008 as the cutoff year for visualizations showing presence of a src directory due to the low number of currently available packages updated prior to 2008. For optimization related dependencies, we used 2009 as the cutoff year and only showed those packages with 15 or greater dependent packages in a given year.

Automatically analyzing software for evidence of optimization has similar difficulties to those mentioned previously related to automatically detecting the use of software testing techniques and tools. The best evidence of software optimization would be in the history of commits, unit tests that time functionality, and package bug reports. While all R packages have source code available, not all have development history available nor unit tests available. Additionally, a stated dependency on one of the optimization packages listed could mean the package creators recommend using that along with their package, not that they are actually using it in their package. Despite these shortcomings, it is estimated that presence of a src directory or the use of specific packages is an indication that some optimization effort was put into a package.

As shown in Figure 3, the evaluation for the presence of a non-empty src directory shows that there is an increasing trend in using compiled code in R packages, by count. However, when evaluated as a percent of all R packages, the change has only been a slight increase over the last few years. Table S3 contains the data used to generate Figure 3.

As shown in Figure 4, Rcpp is the most common optimization related dependency followed by parallel and foreach. Those same packages have been the most popular for packages last updated during the entire period shown. There 641 packages that list dependencies to more than one optimization framework (377 with 2 dependencies, 49 w/4, 15 w/5, 2 w/6, 1 w/7), so the total number of packages shown in the histogram includes some that are double-counted. Table S4 contains the data used to generate Figure 4.

As the numbers from Metric 1 do not match the numbers of Metric 2, some additional exploration is necessary. In terms of total difference, there are 839 more packages using compiled code vs those with one of the searched for dependencies. There are 1706 packages that do not list a dependency to one of the specified packages, but have a src directory for compiled code. There are 867 packages that list a dependency to one of the specified packages but do not have a src directory.

**IDENTIFY AND VALIDATE PERFORMANCE TARGET**

The first step to software optimization is to understand the functional and non-functional requirements of the software being built. Based on expected input, output, and platform the software will be run on, one can make a decision as to what is good enough for the software being developed. A pragmatic approach is best – do not spend time optimizing if it does not add value.

The Python Foundation’s Python language wiki provides excellent high-level advice (*Performance Tips*): First get the software working correctly, test to see if it is correct, profile the application if it is slow, and lastly optimize based on the results of code profiling. If necessary, repeat multiple cycles of testing, profiling, and optimization phases.

Once the functional requirements have been correctly implemented and validated, a decision point is reached: decide if the software is slow and in need of evaluation and optimization. While this may seem a trivial and unnecessary step, it should not be overlooked; a careful evaluation of costs of vs benefit from an optimization effort should be evaluated before moving forward. Some methods for gathering the performance target are through an evaluation of other similar software, interdependencies of the software and its interaction with other systems, and discussion with other experts in the field.

Once a performance target has been identified, development of acceptance tests for performance can begin. While performance testing is often considered an anti-pattern of testing (Jarkko Moilanen) some repeatable tests should be created to track performance as development progresses. Often a ‘stress test’ or a test with greater than expected input/usage is the best way to do this. A good target is to check an order of magnitude larger input than expected. This type of testing can provide valuable insight into the performance characteristics of the software as well unearth potentials for failure due to unexpected load (Sommerville, 2015).

Here is an example of performance validation testing from PCCC:

library(pccc)

rm(list=ls())

gc()

icd10\_large <- feather::read\_feather("../icd\_file\_generator/icd10\_sample\_large.feather")

library(microbenchmark)

microbenchmark(

ccc(icd10\_large[1:10000, c(1:45)], # get id, dx, and pc columns

id = id,

dx\_cols = dplyr::starts\_with("dx"),

pc\_cols = dplyr::starts\_with("pc"),

icdv = 10),

times = 10)

As runs of a software can differ significantly from one run to the next due to other software running on the system, a good starting point is to run the same test 10 times and find the mean value. R’s microbenchmark also shows median, lower and upper quartiles, min, and max. These metrics can help to identify if a test was valid and indicate a need for retesting; i.e. a large interquartile range may indicate not enough tests were run or some aspect of environment is causing performance variations.

Lastly, although the example above is focused on runtime, it is important to also identify targets for disk space used and memory required to complete all desired tasks. There are many optimizations that can be considered to minimize RAM and disk resources.

**BIG O NOTATION**

Big O notation is a method for mathematically determining the upper bound on performance of a block of code without consideration for language and hardware specifics. Although performance can be evaluated in terms of storage or run time, most examples and comparisons focus on run time. However, when working with large datasets, memory usage and disk usage can be of equal or higher importance than run. Big O notation is reported in terms of input (usually denoted as n) and allows one to quickly compare theoretical performance of different algorithms.

The basic steps for evaluating the upper bound of performance of a block of software code is to evaluate what code will run as n approaches infinity. Items that are constant time (regardless of if they run once or x times independent of input) are reduced down to O(1). The key factors that contribute to Big O are loops – a single for loop or similar construct through recursion that runs once for all n is O(n); a nested for loop would be O(n2). Search algorithms that rely on presorted data sets are usually a logarithmic power of n; for example binary search (“Binary search algorithm,” 2018) is O(log2 n). When calculating Big O for a code block, function, or software system, lower order terms are ignored, and just the largest Big O notation is used; for example if a code block is O(1) + O(n) + O(n3) it would be denoted as O(n3).

Despite the value of understanding the theoretical upper bound of software in an ideal situation, there are many difficulties that arise during implementation that can make Big O difficult to calculate and which could make a large Big O faster than a small Big O under actual input conditions. Some key takeaways to temper a mathematical evaluation of Big O are:

* Constants matter when choosing an algorithm - for example if one algorithm is O(56n2), there exists some n where O(n3) is faster.
* Average or best case run time might be more relevant. Although Big O can indicate worst case performance, perhaps average case is more meaningful. Depending on input and operations performed, shortcuts might be implemented to make worst case performance less likely.
* Big O evaluation of algorithms in high level languages are often hard to quantify. In some cases, source code is not available or functions may be calling other functions that were written by someone else. This is best illustrated with an example – without looking at the R or C/C++ code behind each these function calls, it is difficult to know what Big O this should be evaluated to:

if (!missing(id)) {

ids <- dplyr::select(data, !!dplyr::enquo(id))

} else {

ids <- data.frame()[1:nrow(data), ]

}

dplyr::bind\_cols(ids, ccc\_mat\_rcpp(dxmat, pcmat, icdv))

For additional details on Big O notation, see the excellent and broadly understandable introduction to Big O notation by Justin Abrahms (Justin Abrahms).

**CODE PROFILING/BENCHMARKING**

As discussed throughout this section, optimization is a key aspect of software development, especially with respect to large datasets. Although identification of performance targets and a mathematical analysis of algorithms are important steps, the final result must be tested and verified. The only way to know if your software will perform adequately under ideal (and non-ideal) circumstances is to use benchmarking and code profiling tools. Code profilers show how a software behaves and what functions are being called while benchmarking tools generally focus on just execution time – though some tools combine both profiling and benchmarking. In R, some of the common tools are microbenchmark, tictoc, Rprof (R Core Team, 2018), proftools (Tierney & Jarjour, 2016), and profvis.

If, after implementation has been completed, the software functions correctly, and performance targets have not been met, look to optimize your code. Follow an iterative process of profiling to find bottlenecks, making software adjustments, testing small sections with benchmarking and then repeating the process with overall profiling again. If at any point in the process you discover that due to input size, functional requirements, hardware limitations, or software dependencies you cannot make a significant impact to performance, consider stopping further optimization efforts (Burns, 2012).

As with software testing and software bugs, the Pareto principle applies, though some put the balance between code and execution time is closer to 90% of time is in 10% of the code or even as high as 99% in 1% (Jim Xochellis, 2010; Jim Bird, 2013). Identify the biggest bottlenecks via code profiling and focus only on the top issues first. As an example of how to perform code profiling and benchmarking in R, do the following:

First, run profvis to identify the location with the largest execution time:

library(pccc)

icd10\_large <- feather::read\_feather("icd10\_sample\_large.feather")

profvis::profvis({ccc(icd10\_large[1:10000,],

id = id,

dx\_cols = dplyr::starts\_with("dx"),

pc\_cols = dplyr::starts\_with("pc"),

icdv = 10)}, torture = 100)

In Figure 5 you can see a visual depiction of memory allocation, known as a “Flame Graph”, as well as execution time and call stack. By clicking on each item in the stack you will be taken directly to the relevant source code and can see which portions of the code take the most time or memory allocations. Figure 6 is a depiction of the data view which shows just the memory changes, execution time, and source file.

Once the bottleneck has been identified, if possible extract that code to a single function or line that can be run repeatedly with a library such as microbenchmark or tictoc to see if a small change either improves or degrades performance. Test frequently and make sure to compare against previous versions. You may find that something you thought would improve performance degrades performance. As a first step we recommend running tictoc to get general timings such as this:

library(tictoc)

tic("timing: r version")

out <- dplyr::bind\_cols(ids, ccc\_mat\_r(dxmat, pcmat, icdv))

toc()

tic("timing: c++ version")

dplyr::bind\_cols(ids, ccc\_mat\_rcpp(dxmat, pcmat, icdv))

toc()

Then run microbenchmark on more specific sections of code such as this:

library(microbenchmark)

microbenchmark(

lapply(icd10\_small, function(i) {

if(any(stringi::stri\_startswith\_fixed(i, 'S'),na.rm = TRUE))

return(1L)

else

return(0L)

}), 100)

library(stringi)

microbenchmark(

lapply(icd10\_small, function(i) {

if(any(stri\_startswith\_fixed(i, 'S'),na.rm = TRUE))

return(1L)

else

return(0L)

}), 100)

Some of the lessons learned along the way is that in some systems/configurations we found that the use of the “::” operator, as opposed to importing a package, caused a noticeable performance hit. Also widely known and found to be applicable here is that the use of matrices are preferred for performance reasons over data.frame or tibbles. Matrices do have different functionality which can require some re-work when converting from one to another. Another key point found is that an “env” with no parent environment is significantly faster (up to 50x) than one with a parent env. In the end, optimization efforts resulted in reducing run time by 80%.

One limitation with R profiling tools is that if the code to be profiled executes C++ code, you will get no visibility into what is happening once the switch from R to C++ has occurred. As shown in Figure 7, visibility into timing and memory allocation stops at the .Call() function. In order to profile C++ code, you need to use non-R specific tools such as XCode on macOS or gprof on Unix based OSes. See “R\_with\_C++\_profiling.md” in our source code repository for some guidance on this topic.

Some general lessons learned from profiling and benchmarking:

* “Beware the dangers of premature optimization of your code. Your first duty is to create clear, correct code.” (Knuth, 1974; Burns, 2012) Never optimize before you actually know what is taking all the time/memory/space with your software. Different compilers and core language updates often will change or reverse what experience has previously indicated as sources of slowness. Always benchmark and profile before making a change.
* Start development with a high-level programming language first – Developer/Researcher time is more valuable than CPU/GPU time. Choose the language that allows the developer/researcher to rapidly implement the desired functionality rather than selecting a language/framework based on artificial benchmarks.
* Software timing is highly OS, compiler, and system configuration specific. What improves results greatly on one machine and configuration may actually slow performance on another machine. Make sure you test on a range of realistic configurations before deciding that an “improvement” is beneficial.
* If you’ve exhausted your options with your chosen high-level language, C++ is usually the best option for further optimization. For an excellent introduction to combining C++ with R via the library Rcpp, see (Eddelbuettel & Balamuta, 2017).

For some additional information on R optimization, see (Wickham, 2014b; Emily Robinson, 2017).

**CONCLUSION**

Researchers frequently develop software to automate tasks and speed the pace of research. Unfortunately, researchers are rarely trained in software engineering principles necessary to develop robust, validated, and performant software. Software maintenance is an often overlooked and underestimated aspect in the lifecycle of any software product. Software engineering principles and tooling place special focus on the processes around designing, building, and maintaining software. In this paper, the key topics of software testing and software optimization have been discussed along with some analysis of existing software packages in the R language. Our analysis showed that the majority of R packages have neither unit testing nor evidence of optimization available with normally distributed source code. Through self-education on unit testing and optimization, any computational or other researcher can pick up the key principles of software engineering that will enable them to spend less time troubleshooting software and more time doing the research they enjoy.

**REFERENCES**

Agruss C., Johnson B. 2000. Ad Hoc Software Testing. *Viitattu* 4:2009.

Allaire JJ., Francois R., Ushey K., Vandenbrouck G., library MG (TinyThread., http://tinythreadpp.bitsnbites.eu/)., RStudio., library I (Intel T., https://www.threadingbuildingblocks.org/)., Microsoft. 2018. *RcppParallel: Parallel Programming Tools for “Rcpp.”*

Atchison WF., Conte SD., Hamblen JW., Hull TE., Keenan TA., Kehl WB., McCluskey EJ., Navarro SO., Rheinboldt WC., Schweppe EJ., Viavant W., Young DM Jr. 1968. Curriculum 68: Recommendations for Academic Programs in Computer Science: A Report of the ACM Curriculum Committee on Computer Science. *Commun. ACM* 11:151–197. DOI: 10.1145/362929.362976.

Bengtsson H. 2018. *future: Unified Parallel and Distributed Processing in R for Everyone*.

Bengtsson H., R Core Team. 2018. *future.apply: Apply Function to Elements in Parallel using Futures*.

Binary search algorithm 2018. *Wikipedia*.

Bischl B., Lang M. 2015. *parallelMap: Unified Interface to Parallelization Back-Ends*.

Bischl B., Lang M., Mersmann O., Rahnenführer J., Weihs C. 2015. BatchJobs and BatchExperiments: Abstraction Mechanisms for Using R in Batch Environments. *Journal of Statistical Software* 64:1–25.

Burger M., Juenemann K., Koenig T. 2015. *RUnit: R Unit Test Framework*.

Burns P. 2012. *The R Inferno*. lulu.com.

Calaway R., Analytics R., Weston S. 2017. *doMC: Foreach Parallel Adaptor for “parallel.”*

Calaway R., Corporation M., Weston S. 2017. *doSNOW: Foreach Parallel Adaptor for the “snow” Package*.

Calaway R., Corporation M., Weston S., Tenenbaum D. 2018. *doParallel: Foreach Parallel Adaptor for the “parallel” Package*.

Calaway R., Microsoft., Weston S. 2017. *foreach: Provides Foreach Looping Construct for R*.

Chang W., Luraschi J. 2018. *profvis: Interactive Visualizations for Profiling R Code*.

David Heinemeier Hansson 2014.TDD is dead. Long live testing. (DHH). *Available at* *http://david.heinemeierhansson.com/2014/tdd-is-dead-long-live-testing.html*

Dehaghani SMH., Hajrahimi N. 2013. Which Factors Affect Software Projects Maintenance Cost More? *Acta Informatica Medica* 21:63–66. DOI: 10.5455/AIM.2012.21.63-66.

DeWitt P., Bennett T., Feinstein J., Russell S. 2017. *pccc: Pediatric Complex Chronic Conditions*.

Dragulescu AA., Arendt C. 2018. *xlsx: Read, Write, Format Excel 2007 and Excel 97/2000/XP/2003 Files*.

Eckert A. 2018. *parallelDist: Parallel Distance Matrix Computation using Multiple Threads*.

Eddelbuettel D., Balamuta JJ. 2017. *Extending R with C++: A Brief Introduction to Rcpp*. PeerJ Inc. DOI: 10.7287/peerj.preprints.3188v1.

Emily Robinson 2017.Making R Code Faster : A Case Study. *Available at* *https://robinsones.github.io/Making-R-Code-Faster-A-Case-Study/*

Feinerer I., Theussl S., Buchta C. 2015. *DSL: Distributed Storage and List*.

Feudtner C., Christakis DA., Connell FA. 2000. Pediatric Deaths Attributable to Complex Chronic Conditions: A Population-Based Study of Washington State, 1980–1997. *Pediatrics* 106:205–209.

Feudtner C., Feinstein JA., Zhong W., Hall M., Dai D. 2014. Pediatric complex chronic conditions classification system version 2: updated for ICD-10 and complex medical technology dependence and transplantation. *BMC Pediatrics* 14:199. DOI: 10.1186/1471-2431-14-199.

Gaslam B. 2017. *unitizer: Interactive R Unit Tests*.

Glass RL. 2001. Frequently Forgotten Fundamental Facts About Software Engineering. *IEEE Softw.* 18:112–111. DOI: 10.1109/MS.2001.922739.

Grosjean P. 2014. *SciViews-R: A GUI API for R*. MONS, Belgium: UMONS.

Izrailev S. 2014. *tictoc: Functions for timing R scripts, as well as implementations of Stack and List structures.*

James Carr 2015.TDD Anti-Patterns. *Available at* *https://web.archive.org/web/20150726134212/http://blog.james-carr.org:80/2006/11/03/tdd-anti-patterns/*

Jarkko Moilanen. *Test Driven Development details*.

Jim Bird 2013.Applying the 80:20 Rule in Software Development - DZone Agile. *Available at* *https://dzone.com/articles/applying-8020-rule-software*

Jim Xochellis 2010.The impact of the Pareto principle in optimization - CodeProject. *Available at* *https://www.codeproject.com/Articles/49023/The-impact-of-the-Pareto-principle-in-optimization*

jpreese.Best practices for writing unit tests. *Available at* *https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices* (accessed August 28, 2018).

Justin AbrahmsBig-O notation explained by a self-taught programmer. *Available at* *https://justin.abrah.ms/computer-science/big-o-notation-explained.html*

Kent Beck, Erich Gamma 1998. Test Infected: Programmers Love Writing Tests. *Java Report* 3.

Knuth DE. 1974. Structured Programming with Go to Statements. *ACM Comput. Surv.* 6:261–301. DOI: 10.1145/356635.356640.

Koskinen, Jussi. 2015. Software Maintenance Costs

Kusnierczyk W., Eddelbuettel D., Hasselman B. 2012. *rbenchmark: Benchmarking routine for R*.

Leek JT., Peng RD. 2015. Opinion: Reproducible research can still be wrong: Adopting a prevention approach. *Proceedings of the National Academy of Sciences* 112:1645–1646. DOI: 10.1073/pnas.1421412111.

Lentin J., Hennessey A. 2017. *unittest: TAP-Compliant Unit Testing*.

Luraschi J., Kuo K., Ushey K., Allaire JJ., Macedo S., RStudio., Foundation TAS. 2018. *sparklyr: R Interface to Apache Spark*.

Matloff N. 2016. Software Alchemy: Turning Complex Statistical Computations into Embarrassingly-Parallel Ones. *Journal of Statistical Software* 71:1–15. DOI: 10.18637/jss.v071.i04.

Mersmann O. 2018. *microbenchmark: Accurate Timing Functions*.

Nolan R., Padilla-Parra S. 2017. exampletestr—An easy start to unit testing R packages. *Wellcome Open Research* 2. DOI: 10.12688/wellcomeopenres.11635.2.

Nutter B., Lane S. 2018. *redcapAPI: Accessing data from REDCap projects using the API*. DOI: 10.5281/zenodo.11826.

*Performance Tips*

R Core Team 2018. *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing.

*Ranorex* Ranorex GmbH.

Rooney P. 2002.Microsoft’s CEO: 80-20 Rule Applies To Bugs, Not Just Features. *Available at* *http://www.crn.com/news/security/18821726/microsofts-ceo-80-20-rule-applies-to-bugs-not-just-features.htm*

*Selenium*

Sommerville I. 2015. *Software Engineering*. Boston: Pearson.

Sommerville I. 2016.Giving up on test-first development. *Available at* *http://iansommerville.com/systems-software-and-technology/giving-up-on-test-first-development/*

SparkR (R on Spark) - Spark 2.3.2 Documentation. *Available at* *https://spark.apache.org/docs/latest/sparkr.html*

SUnit 2017. *Wikipedia*.

The Joint Task Force on Computing Curricula 2015. *Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering*. New York, NY, USA: ACM.

Tierney L., Jarjour R. 2016. *proftools: Profile Output Processing Tools for R*.

Tierney L., Rossini AJ., Li N., Sevcikova H. 2018. *snow: Simple Network of Workstations*.

*Unified Functional Testing* Microfocus.

*Unit testing Anti-patterns catalogue*

Weston S. 2017. *doMPI: Foreach Parallel Adaptor for the Rmpi Package*.

Wickham H. 2011. testthat: Get Started with Testing. *The R Journal* 3:5–10.

Wickham H. 2014a. *profr: An alternative display for profiling information*.

Wickham H. 2014b. *Advanced R*. Boca Raton, FL: Chapman and Hall/CRC.

Wilson G. 2016. Software Carpentry: lessons learned. *F1000Research*. DOI: 10.12688/f1000research.3-62.v2.

Wilson G., Aruliah DA., Brown CT., Hong NPC., Davis M., Guy RT., Haddock SHD., Huff KD., Mitchell IM., Plumbley MD., Waugh B., White EP., Wilson P. 2014. Best Practices for Scientific Computing. *PLOS Biology* 12:e1001745. DOI: 10.1371/journal.pbio.1001745.

Xie Y. 2015. *Dynamic Documents with R and knitr*. Boca Raton: Chapman and Hall/CRC.

Xie Y. 2018. *testit: A Simple Package for Testing R Packages*.

xUnit 2017. *Wikipedia*.

Yu H. 2002. Rmpi: Parallel Statistical Computing in R. *R News* 2:10–14.