Software engineering principles to improve quality and performance of R software

Seth Russell[1], Tellen D. Bennett[1,2], Debashis Ghosh[1,3]

[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

INTRODUCTION

Writing scientific software has progressed from the work of early pioneers to a range of computer professionals, computational researchers, and self-taught individuals. The educational discipline of computer science, standardized many years ago through recommendations from the Association for Computing Machinery (ACM) (Atchison et al., 1968), has grown in breadth and depth over many years. Software engineering, a discipline within computer science, "seeks to develop and use systematic models and reliable techniques to produce high-quality software. These software engineering concerns extend from theory and principles to development practices that are most visible to those outside the discipline." (The Joint Task Force on Computing Curricula, 2015)

As they gain sophistication, computational researchers, statisticians, and similar professionals need to advance their skills by adopting principles of software engineering. Wilson et al. identified 8 key areas where scientists can benefit from software engineering best practices (Wilson et al., 2014). The term "best" as referenced in the previous cited work and others cited later refers to expert consensus based on knowledge and observational reporting of results from application of the practices. They provide a high-level description of 8 important

40 principles of software engineering that should "reduce the number of errors in scientific
41 software, make it easier to reuse, and save the authors of the software time and effort that can
42 used for focusing on the underlying scientific questions." While their principles are still relevant
43 and important today, there has not been enough progress in this endeavor, especially with respect
44 to software testing and software optimization principles (Wilson, 2016; Nolan & Padilla-Parra,
45 2017).
46
47 The ACM/Institute of Electrical and Electronics Engineers (IEEE) recommendations for an
48 undergraduate degree in software engineering describe a range of coursework and learning
49 objectives. Their guidelines call out 10 specific knowledge areas that should be part of or guide
50 all software engineering coursework. The major areas are: computer science fundamentals, math
51 and engineering fundamentals, professional interactions/communication, software modeling,
52 requirement gathering, software design, verification, project processes, quality, and security (The
53 Joint Task Force on Computing Curricula, 2015). These major themes are not covered
54 extensively outside software engineering and include such generally applicable items such as
55 software verification, validation, testing, and computer science fundamentals (e.g. software
56 optimization, modeling, and requirement gathering).
57
58 In addition to the need for further training, understanding the software lifecycle is necessary: the
59 process of software development from ideation to delivery of code. The largest component of
60 software's lifecycle is maintenance. Software maintenance costs are large and increasing (Glass,
61 2001; Dehaghani & Hajrahimi, 2013; Koskinen, 2015); some put maintenance at 90% of total
62 software cost. The chief factor in software maintenance cost is the time of the people creating
63 and using the software. From the recent trend on making research results reproducible and
64 replicable, some recommend making code openly available to any who might wish to repeat or
65 further analyze results (Leek & Peng, 2015). With the development of any software artifact, an
66 important consideration for implementation should be maintenance. As research scientists tend to
67 think of their software products as unique tools that will not be used regularly or for a long
68 period, they often do not consider long term maintenance issues during the development phase
69 (Sandve et al., 2013; Prins et al., 2015). While a rigorous and formal software engineering
70 approach is not well suited to the standard lifecycle of research software (Wilson, 2016), there
71 are many techniques that can help to reduce cost of maintenance and speed development. While
72 best practices such as the use of version control software, open access to data, software, and
73 results are becoming more wide spread, other practices such as testing and optimization need
74 further attention.
75
76 In this paper, a brief survey of currently available R packages from The Comprehensive R
77 Archive Network (CRAN) will be used to show the continued need for software testing and
78 optimization. Source code for this analysis is freely available at https://github.com/magic-
79 lantern/SoftwareEngineeringPrinciples. After the presentation of the current state of R packages,

80   general advice on software testing and optimization will be presented. The R package "pccc:
81   Pediatric Complex Chronic Conditions" (Feinstein et al., 2018; DeWitt et al., 2017) (`pccc`),
82   available via CRAN and at https://github.com/CUD2V/pccc, is used for code examples in this
83   article. `pccc` is a combined R and C++ implementation of the Pediatric Complex Chronic
84   Conditions software released as part of a series of research papers (Feudtner, Christakis &
85   Connell, 2000; Feudtner et al., 2014). `pccc` takes as input a data set containing International
86   Statistical Classification of Diseases and Related Health Problems (ICD) Ninth revision or Tenth
87   revision diagnosis and procedure codes and outputs which if any complex chronic conditions a
88   patient has.
89

ANALYSIS OF R PACKAGES ON CRAN

91

TESTING OF R PACKAGES

93   In order to estimate the level of testing common among R software, we analyzed all R packages
94   available through CRAN. Although Nolan (Nolan & Padilla-Parra, 2017) performed a similar
95   analysis in the past, due to the rapid change in the CRAN as a whole, a reevaluation is necessary.
96   At the time of Nolan's work, CRAN contained 10084 packages; it now contains 13509.
97   Furthermore, the analysis by Nolan had a few shortcomings that we have addressed in this
98   analysis: there are additional testing frameworks for which we wanted to analyze their usage; not
99   all testing frameworks and R packages store their test code in a directory named "tests"; only
100   packages modified in the past 2 years were reported - there are many commonly used R packages
101   that have not been updated in the last 2 years.
102

103   Although we address some shortcomings in analyzing R code for use of testing best practices,
104   our choice of domain for analysis does have some limitations. Not all research software is
105   written in R; for those that do use R, not all software development results in a package published
106   on CRAN. While other software languages have tools for testing, additional research would be
107   needed to evaluate level of testing in those languages to see how it compares to this analysis.
108   Although care has been taken to identify standard testing use cases and practices for R, testing
109   can be performed in-line through use of core functions such as `stop()` or `stopifnot()`. Also,
110   developers may have their own test cases they run while developing their software, but did not
111   include them in the package made available on CRAN. Unit tests can be considered executable
112   documentation, a key method of conveying how to use software correctly (Reese, 2018).
113   Published research that involves software is not as easy to access and evaluate for use of testing
114   code as CRAN packages are. While some journals have standardized means for storing and
115   sharing code, many leave the storing and sharing of code up to the individual author, creating an
116   environment where code analysis would require significant manual effort.
117

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

120

121   Metric 1: In the source code of each package, search for non-empty testing directories
122   using the regular expression pattern "[Tt]est[^/]*/.+". All commonly used R testing
123   packages (those identified for metric 2) recommend placing tests in a directory by
124   themselves, which we look for.

125

126   Metric 2: Check for stated dependencies on one of the following testing packages: `RUnit`
127   (Burger, Juenemann & Koenig, 2015), `svUnit` (Grosjean, 2014), `testit` (Xie, 2018),
128   `testthat` (Wickham, 2011), `unitizer` (Gaslam, 2017), or `unittest` (Lentin &
129   Hennessey, 2017). From the authors of these packages, it is recommended to list
130   dependency (or dependencies) to a testing framework even though standard usage of a
131   package may not require it.

132

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

135

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

139

140 As shown in Figure 2, reliance upon testing frameworks is increasing over time both in count
141 and as a percentage of all packages. There 16 packages that list dependencies on more than one
142 testing framework (9 with dependencies on both `RUnit` and `testthat`, 7 with dependencies on
143 both `testit` and `testthat`), so the total number of packages shown in the histogram includes
144 16 that are double counted. Table S2 contains the data used to generate Figure 2.

145

146 As the numbers from Metric 1 do not match the numbers of Metric 2, some additional
147 exploration is necessary. There are 884 more packages identified from Metric 1 vs Metric 2.
148 There are 1115 packages that do not list a dependency to a testing framework, but have a testing
149 directory; e.g. the package `xlsx` (Dragulescu & Arendt, 2018). Some packages use a testing
150 framework, but do not list it as a dependency; e.g. the package `redcapAPI` (Nutter & Lane,
151 2018). There are also 231 packages that list a testing framework as a dependency, but do not
152 contain a directory with tests. See Supplemental tables S1 and S2 for more details.

153

154 **OPTIMIZATION OF R PACKAGES**
155 In order to estimate the level of software optimization common among R software, we performed
156 an analysis of all R packages available through CRAN. To analyze the use of software
157 optimization tools and techniques, we evaluated all CRAN packages on two different metrics:

158

159   Metric 1: In the source code of each package, search for non-empty src directories using
160   the regular expression pattern "src[^/]*/.+". By convention, packages using compiled
161   code (C, C++, Fortran) place those files in a '/src' directory.
162
163   Metric 2: Check for stated dependencies on packages that can optimize, scale
164   performance, or evaluate performance of a package. Packages included in analysis are:
165   DSL (Feinerer, Theussl & Buchta, 2015), Rcpp (Eddelbuettel & Balamuta, 2017),
166   RcppParallel (Allaire et al., 2018a), Rmpi (Yu, 2002), SparkR (Apache Software
167   Foundation, 2018), batchtools (Bischl et al., 2015), bench (Hester, 2018), benchr
168   (Klevtsov, Antonov & Upravitelev, 2018), doMC (Calaway, Analytics & Weston, 2017),
169   doMPI (Weston, 2017), doParallel (Calaway et al., 2018), doSNOW (Calaway,
170   Corporation & Weston, 2017), foreach (Calaway, Microsoft & Weston, 2017), future
171   (Bengtsson, 2018), future.apply (Bengtsson & R Core Team, 2018),
172   microbenchmark (Mersmann, 2018), parallel (R Core Team, 2018),
173   parallelDist (Eckert, 2018), parallelMap (Bischl & Lang, 2015), partools
174   (Matloff, 2016), profr (Wickham, 2014a), profvis (Chang & Luraschi, 2018),
175   rbenchmark (Kusnierczyk, Eddelbuettel & Hasselman, 2012), snow (Tierney et al.,
176   2018), sparklyr (Luraschi et al., 2018), tictoc (Izrailev, 2014).
177
178 For the optimization analysis, we used 2008 as the cutoff year for visualizations showing
179 presence of a src directory due to the low number of currently available packages last updated
180 prior to 2008. For optimization related dependencies, in order to aid visual understanding, we
181 used 2009 as the cutoff year and only showed those packages with 15 or greater dependent
182 packages in a given year.
183
184 Automatically analyzing software for evidence of optimization has similar difficulties to those
185 mentioned previously related to automatically detecting the use of software testing techniques
186 and tools. The best evidence of software optimization would be in the history of commits, unit
187 tests that time functionality, and package bug reports. While all R packages have source code
188 available, not all have development history available nor unit tests available. Additionally, a
189 stated dependency on one of the optimization packages listed could mean the package creators
190 recommend using that along with their package, not that they are actually using it in their
191 package. Despite these shortcomings, it is estimated that presence of a src directory or the use of
192 specific packages is an indication that some optimization effort was put into a package.
193
194 As shown in Figure 3, the evaluation for the presence of a non-empty src directory shows that
195 there is an increasing trend in using compiled code in R packages, by count. However, when
196 evaluated as a percent of all R packages, the change has only been a slight increase over the last
197 few years. Table S3 contains the data used to generate Figure 3.
198

199   As shown in Figure 4, in 2018, `Rcpp` is the most common optimization related dependency
200   followed by `parallel` and `foreach`. Those same packages have been the most popular for
201   packages last updated during the entire period shown. There 699 packages that list dependencies
202   to more than one optimization framework (407 with 2 dependencies, 220 w/3, 53 w/4, 16 w/5, 2
203   w/6, 1 w/7), so the total number of packages shown in the histogram includes some that are
204   double-counted. Table S4 contains the data used to generate Figure 4.
205
206   As the numbers from Metric 1 do not match the numbers of Metric 2, some additional
207   exploration is necessary. In terms of total difference, there are 818 more packages using
208   compiled code vs those with one of the searched for dependencies. There are 1726 packages that
209   do not list a dependency to one of the specified packages, but have a src directory for compiled
210   code. There are 908 packages that list a dependency to one of the specified packages but do not
211   have a src directory. See Supplemental tables S3 and S4 for more details.
212
213   # RECOMMENDATIONS TO IMPROVE QUALITY AND PERFORMANCE
214
215   ## SOFTWARE TESTING
216
217   Whenever software is written as part of a research project, careful consideration should be given
218   to how to verify that the software performs the desired functionality and produces the desired
219   output. As with bench science, software can often have unexpected and unintended results due to
220   minor or even major problems during the implementation process. Software testing is a well-
221   established component of any software development lifecycle (Atchison et al., 1968) and should
222   also be a key component of research software. As shown previously, even among R software
223   packages intended to be shared with and used by others, the majority of R packages (67% to 73%
224   depending on metric) do not have tests that are made available with the package.
225
226   Various methodologies and strategies exist for software testing and validation as well as how to
227   integrate software with a software development lifecycle. Some common testing strategies are no
228   strategy, ad hoc testing (Agruss & Johnson, 2000), test driven development (TDD) (Beck &
229   Gamma, 1998). There are also common project methodologies where testing fits into the project
230   lifecycle; two common examples are the waterfall project management methodology, where
231   testing is a major phase that occurs at a specific point in time, and the agile project management
232   methodology (Beck et al., 2001), where there are many small iterations including testing. While
233   a full discussion of various methods and strategies is beyond the scope of this article, three key
234   concepts presented are: when to start testing, what to test, and how to test.
235
236   **Key recommendations for when to test:**
237   • Build tests before implementation
238   • Test after functionality has been implemented

239

240   **Discussion:** One of the popular movements in recent years has been to develop tests first and
241   then implement code to meet desired functionality, a strategy called TDD. While the TDD
242   strategy has done much to improve the focus of the software engineering world on testing, some
243   have found that it does not work with all development styles (Hansson, 2014; Sommerville,
244   2016), and others have reported that it does not increase developer productivity, reduce overall
245   testing effort, nor improve code quality in comparison to other testing methodologies (Fucci et
246   al., 2016). An approach that more closely matches the theoretically based software development
247   cycle and flexible nature of research software is to create tests after a requirement or feature has
248   been implemented (Osborne et al., 2014; Kanewala & Bieman, 2014). As developing
249   comprehensive tests of software functionality can be a large burden to accrue at a single point in
250   time, a more pragmatic approach is to alternate between developing new functionality and
251   designing tests to validate new functionality. Similar to the agile software development strategy,
252   a build/test cycle can allow for quick cycles of validated functionality that help to provide input
253   into additional phases of the software lifecycle.

254

255   **Key recommendations for what to test:**
256   • Identify the most important or unique feature(s) of software being implemented. Software
257   bugs are found to follow a Pareto or Zipfian distribution.
258   • Test data and software configuration
259   • If performance is a key feature, build tests to evaluate performance.

260

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

269

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

274

275   Once answers to these questions are known, the developer(s) should spend time designing tests
276   to validate key features, avoiding major negatives, and ensuring software performs adequately.
277   Optimization and performance recommendations are covered in the "Software Optimization"
278   section. Part of the test design process should include how to "test" more than just the code.

279 Some specific aspects of non-code tests include validation of approach and implementation
280 choices with a mentor or colleague.
281
282 As a brief example of how to apply the aforementioned testing principles, we provide some
283 information on testing steps followed during the `pccc` package development process. The first
284 tests written were those that were manually developed and manually run as development
285 progressed. Key test cases of this form are ideal candidates for inclusion in automated testing.
286 The first tests were taking a known data set, running our process to identify how many of the
287 input rows had complex chronic conditions, and then report on the total percentages found; this
288 result was then compared with published values.
289

```
290     # read in HCUP KID 2009 Database
291     kid9cols <- read_csv("KID09_core_columns.csv")
292     kid9core <- read_fwf("KID_2009_Core.ASC",
293                 fwf_positions(
294                     start = kid9cols$start,
295                     end = kid9cols$end,
296                     col_names = tolower(kid9cols$name)),
297                 col_types = paste(rep("c", nrow(kid9cols)),
298                     collapse = ""))
299
300     # Output some summary information for manual inspection
301     table(kid9core$year)
302     dim(kid9core)
303     n_distinct(kid9core$recnum)
304
305     # Run process to identify complex chronic conditions
306     kid_ccc <-
307         ccc(kid9core[, c(2, 24:48, 74:77, 106:120)],
308             id      = recnum,
309             dx_cols = vars(starts_with("dx"), starts_with("ecode")),
310             pc_cols = vars(starts_with("pr")),
311             icdv    = 09)
312
313     # Output results for manual inspection
314     kid_ccc
315
316     # Create summary statistics to compare to published values
317     dplyr::summarize_at(kid_ccc, vars(-recnum), sum)
318         %>% print.data.frame
```

```
319    dplyr::summarize_at(kid_ccc, vars(-recnum), mean)
320        %>% print.data.frame
321
```

322 For the `pccc` package there is a large set of ICD codes and code set patterns that are used to
323 determine if an input record meets any complex chronic condition criteria. To validate the
324 correct functioning of the software, we needed to validate the ICD code groupings were correct
325 and were mutually exclusive (as appropriate). As `pccc` is a re-implementation of existing SAS
326 and Stata code, we needed to validate that the codes from the previously developed and
327 published software applications were identical and were performing as expected. Through a
328 combination of manual review and automated comparison codes were checked to see if
329 duplicates and overlaps existed. Any software dealing with input validation or having a large
330 amount of built-in values used for key functionality should follow a similar data validation
331 process.
332
333 As an example of configuration testing, here is a brief snippet of some of the code used to
334 automatically find duplicates and codes that were already included as part of another code:
335

```
336    icds <- input.file("../pccc_validation/icd10_codes_r.txt")
337
338    unlist(lapply(icds, function(i) {
339      tmp <- icds[icds != i]
340      output <- tmp[grepl(paste0("^", i, ".*"), tmp)]
341      # add the matched element into the output
342      if(length(output) != 0)
343        output <- c(i, output)
344      output
345    }))
346
347
```

348 **Key recommendations for how to test:**
349 • Software developer develops unit tests
350 • Intended user of software should perform validation/acceptance tests
351 • Run all tests regularly
352 • Review key algorithms with domain experts.
353

354 **Discussion:** Most programming languages have a multitude of testing tools and frameworks
355 available for assisting developers with the process of testing software. Due to the recurring
356 patterns common across programming languages most languages have a SUnit (Wikipedia
357 contributors, 2017a) derived testing tool, commonly referred to as an "xUnit" (Wikipedia
358 contributors, 2017b) testing framework that focuses on validating individual units of code along

359   with necessary input and output meet desired requirements. Based on software language used,
360   unit tests may be at the class or function/procedure level. Some common xUnit style packages in
361   R are `RUnit` and `testthat`. Unit tests should be automated and run regularly to ensure errors
362   are caught and addressed quickly. For R, it is easy to integrate unit tests into the package build
363   process, but other approaches such as post-commit hook in a version control system are also
364   common.
365

366   In addition to unit tests, typically written by the developers of the software, users should perform
367   acceptance tests, or high-level functionality tests that validate the software meets requirements.
368   Due to the high-level nature and subjective focus of acceptance tests, they are often manually
369   performed and may not follow a regimented series of steps. Careful documentation of how a user
370   will actually use software, referred to as user stories, are translated into step by step tests that a
371   human follows to validate the software works as expected. A few examples of acceptance testing
372   tools that primarily focus GUI aspects of software are: Selenium (Selenium Contributors, 2018),
373   Microfocus Unified Functional Testing (formely known as HP's QuickTest Professional) (Micro
374   Focus, 2018), and Ranorex (Ranorex GmbH, 2018). As research focused software often does not
375   have a GUI, one aide to manual testing processes is for developers of the software or expert users
376   to create a full step by step example via an R Markdown (Allaire et al., 2018b; Xie, Allaire &
377   Grolemund, 2018) notebook demonstrating use of the software followed by either manually or
378   automatic validation that the expected end result is correct.
379

380   In addition to the tool-based approaches already mentioned, other harder to test items such as
381   algorithms and solution approach should be scrutinized as well. While automated tests can
382   validate mathematical operations or other logic steps are correct, they cannot verify that the
383   approach or assumptions implied through software operations are correct. This level of testing
384   can be done through code review and design review sessions with others who have knowledge of
385   the domain or a related domain.
386

387   During development of the `pccc` package, after the initial tests shown in previous sections,
388   further thought went into how the specifics of the desired functionality should perform. Unit tests
389   were developed to validate core functionality. We also spent time thinking about how the
390   software might behave if the input data was incorrect or if parameters were not specified
391   correctly. If an issue is discovered at this point, a common pattern is to create a test case for
392   discovered bugs that are fixed - this ensures that a re-occurrence, known as a "regression" to
393   software engineers, of this error does not happen again. In the case of `pccc`, developers expected
394   large input comprised of many observations with many variables. When a tester accidentally just
395   passed 1 observation with many variables, the program crashed. The problem was discovered to
396   be due to the flexible nature of the `sapply()` function returning different data types based on
397   input.
398

399   The original code from ccc.R:

400

```
401        # check if call didn't specify specific diagnosis columns
402        if (!missing(dx_cols)) {
403            # assume columns are referenced by 'dx_cols'
404            dxmat <- sapply(dplyr::select(
405                data, !!dplyr::enquo(dx_cols)), as.character)
406            # create empty matrix if necessary
407            if(! is.matrix(dxmat)) {
408                dxmat <- as.matrix(dxmat)
409            }
410        } else {
411
412            dxmat <- matrix("", nrow = nrow(data))
413        }
```

414

415   The new code:

416

```
417        if (!missing(dx_cols)) {
418            dxmat <- as.matrix(dplyr::mutate_all(
419                dplyr::select(
420                    data, !!dplyr::enquo(dx_cols)),
421                as.character))
422        } else {
423            dxmat <- matrix("", nrow = nrow(data))
424        }
```

425

426   One of the tests written to verify the problem didn't reoccur:

427

```
428        # Due to previous use of sapply in ccc.R, this would fail
429        test_that(paste("1 patient with multiple rows of no diagnosis",
430                        "data - should have all CCCs as FALSE"), {
431          expect_true(all(ccc(dplyr::data_frame(
432              id = 'a',
433              dx1 = NA,
434              dx2 = NA),
435            dx_cols = dplyr::starts_with("dx"),
436            icdv   = code) == 0))
437        }
438      )
```

439

440 **Testing Anti-Patterns:** While the above guidance should help researchers know the basics of
441 testing, it does not cover in detail what not to do. An excellent collection of testing anti-patterns
442 can be found at (Moilanen, 2014; Carr, 2015; Stack Overflow Contributors, 2017). Some key
443 problems that novices experience when learning how to test software are:

444

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

457

458 **SOFTWARE OPTIMIZATION**

459

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

467

468 The survey of R packages at the beginning of this work indicates that approximately 75% of
469 packages do not use optimization related packages nor compiled code to improve performance.
470 While the survey of R packages is not evidence of non-optimization of packages in CRAN,
471 computational researchers can should carefully consider performance aspects of their software
472 before declaring it complete. This section will provide a starting point for additional study,
473 research, and experimentation. The Python Foundation's Python language wiki provides
474 excellent high-level advice (Python Wiki Contributors, 2018) to follow before spending too
475 much time in optimization: First get the software working correctly, test to see if it is correct,
476 profile the application if it is slow, and lastly optimize based on the results of code profiling. If
477 necessary, repeat multiple cycles of testing, profiling, and optimization phases. The key aspects

478   of software optimization discussed in this are: identify a performance target, understanding and
479   applying Big O notation, and the use code profiling and benchmarking tools.
480

481   **Key recommendations for identifying and validating performance targets:**
482   •   Identify functional and non-functional requirements of the software being developed.
483   •   If software performance is key to the software requirements, develop repeatable tests to
484   evaluate performance
485

486   **Discussion:** The first step to software optimization is to understand the functional and non-
487   functional requirements of the software being built. Based on expected input, output, and
488   platform the software will be run on, one can make a decision as to what is good enough for the
489   software being developed. A pragmatic approach is best – do not spend time optimizing if it
490   does not add value. Once the functional requirements have been correctly implemented and
491   validated, a decision point is reached: decide if the software is slow and in need of evaluation
492   and optimization. While this may seem a trivial and unnecessary step, it should not be
493   overlooked; a careful evaluation of costs versus benefit from an optimization effort should be
494   evaluated before moving forward. Some methods for gathering the performance target are
495   through an evaluation of other similar software, interdependencies of the software and its
496   interaction with other systems, and discussion with other experts in the field.
497

498   Once a performance target has been identified, development of tests for performance can begin.
499   While performance testing is often considered an anti-pattern of testing (Moilanen, 2014) some
500   repeatable tests should be created to track performance as development progresses. Often a
501   'stress test' or a test with greater than expected input/usage is the best way to do this. A good
502   target is to check an order of magnitude larger input than expected. This type of testing can
503   provide valuable insight into the performance characteristics of the software as well unearth
504   potentials for failure due to unexpected load (Sommerville, 2015).
505

506   Here is an example of performance validation testing that can also serve as a basic
507   reproducibility test calling the main function from `pccc` using the `microbenchmark` package
508   (one could also use `bench`, `benchr`, or other similar R packages).
509

```
510    library(pccc)
511    rm(list=ls())
512    gc()
513
514    icd10_large <-
515      feather::read_feather(
516        "../icd_file_generator/icd10_sample_large.feather"
517        )
518    library(microbenchmark)
```

```
519    microbenchmark(
520    ccc(icd10_large[1:10000, c(1:45)], # get id, dx, and pc columns
521                   id      = id,
522                   dx_cols = dplyr::starts_with("dx"),
523                   pc_cols = dplyr::starts_with("pc"),
524                   icdv    = 10),
525    times = 10)
526
527    Unit: seconds
528     expr      min       lq     mean   median       uq      max neval
529      ccc  2.857625 2.908964 2.959805 2.920408 3.023602 3.119937    10
530
```

Results are from a system with 3.1 GHz Intel Core i7, 16 GB 2133 MHz LPDDR3, PCI-Express SSD, running macOS 10.12.6 and R version 3.5.1 (2018-07-02).

As software runs can differ significantly from one to the next due to other software running on the test system, a good starting point is to run the same test 10 times (rather than the microbenchmark default of 100 due to this being a longer running process) and record the mean run time. microbenchmark also shows median, lower and upper quartiles, min, and max run times. The actual `ccc()` call specifics are un-important; the key is to test the main features of your software in a repeatable fashion and watch for performance changes over time. 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. Software benchmarking is highly system specific in that changing OS version, R version, R dependent package version, compiler version (if compiled code involved), or hardware may change the results. As long as all tests are run the same on the same system with the same software, one can compare timings as development progresses.

Lastly, although the example above is focused on runtime, it can be beneficial to also identify targets for disk space used and memory required to complete all desired tasks. As an example, tools such as `bench` and `profvis` demonstrated in our 'Code Profiling/Benchmarking' section as well as `object.size()` from core R can give developers insight into memory allocation and usage. There are many resources beyond this work that can provide guidance on how to minimize RAM and disk resources (Kane, Emerson & Weston, 2013; Wickham, 2014b; Wickham et al., 2016; Klik, Collet & Facebook, 2018).

Key recommendations for identifying upper bound on performance:
- Big O notation allows the comparison of theoretical performance of different algorithms
- Evaluate how many times blocks of code will run as input approaches infinity
- Loops inside loops are very slow as input approaches infinity

559
560   **Discussion:** Big O notation is a method for mathematically determining the upper bound on
561   performance of a block of code without consideration for language and hardware specifics.
562   Although performance can be evaluated in terms of storage or run time, most examples and
563   comparisons focus on run time. However, when working with large datasets, memory usage and
564   disk usage can be of equal or higher importance than run. Big O notation is reported in terms of
565   input (usually denoted as n) and allows one to quickly compare theoretical performance of
566   different algorithms.
567
568   The basic steps for evaluating the upper bound of performance of a block of software code is to
569   evaluate what code will run as n approaches infinity. Items that are constant time (regardless of if
570   they run once or x times independent of input) are reduced down to $O(1)$. The key factors that
571   contribute to Big O are loops – a single for loop or similar construct through recursion that runs
572   once for all n is $O(n)$; a nested for loop would be $O(n^2)$. When calculating Big O for a code
573   block, function, or software system, lower order terms are ignored, and just the largest Big O
574   notation is used; for example if a code block is $O(1) + O(n) + O(n^3)$ it would be denoted as
575   $O(n^3)$.
576
577   Despite the value of understanding the theoretical upper bound of software in an ideal situation,
578   there are many difficulties that arise during implementation that can make Big O difficult to
579   calculate and which could make a large Big O faster than a small Big O under actual input
580   conditions. Some key takeaways to temper a mathematical evaluation of Big O are:
581
- Constants matter when choosing an algorithm - for example if one algorithm is $O(56n^2)$,
  there exists some n where $O(n^3)$ is faster.
- Average or best case run time might be more relevant.
- Big O evaluation of algorithms in high level languages is often hard to quantify.

587   For additional details on Big O notation, see the excellent and broadly understandable
588   introduction to Big O notation (Abrahms, 2016).
589
590   **Key recommendations for profiling and benchmarking:**
- Profile code to find bottlenecks
- Modify code to address largest items from profiling
- Run tests to make sure functionality isn't affected
- Repeat process if gains are made and additional performance improvements are
  necessary.

596
597   **Discussion:** As discussed throughout this section, optimization is a key aspect of software
598   development, especially with respect to large datasets. Although identification of performance

599 targets and a mathematical analysis of algorithms are important steps, the final result must be
600 tested and verified. The only way to know if your software will perform adequately under ideal
601 (and non-ideal) circumstances is to use benchmarking and code profiling tools. Code profilers
602 show how a software behaves and what functions are being called while benchmarking tools
603 generally focus on just execution time – though some tools combine both profiling and
604 benchmarking. In R, some of the common tools are `bench`, `benchr`, `microbenchmark`,
605 `tictoc`, `Rprof` (R Core Team, 2018), `proftools` (Tierney & Jarjour, 2016), and `profvis`.
606
607 If, after implementation has been completed, the software functions correctly, and performance
608 targets have not been met, look to optimize your code. Follow an iterative process of profiling to
609 find bottlenecks, making software adjustments, testing small sections with benchmarking and
610 then repeating the process with overall profiling again. If at any point in the process you discover
611 that due to input size, functional requirements, hardware limitations, or software dependencies
612 you cannot make a significant impact to performance, consider stopping further optimization
613 efforts (Burns, 2012).
614
615 As with software testing and software bugs, the Pareto principle applies, though some put the
616 balance between code and execution time is closer to 90% of time is in 10% of the code or even
617 as high as 99% in 1% (Xochellis, 2010; Bird, 2013). Identify the biggest bottlenecks via code
618 profiling and focus only on the top issues first. As an example of how to perform code profiling
619 and benchmarking in R, do the following:
620
621 First, use `profvis` to identify the location with the largest execution time:
622

```
623      library(pccc)
624      icd10_large <-
625      feather::read_feather("icd10_sample_large.feather")
626      profvis::profvis({ccc(icd10_large[1:10000,],
627                          id      = id,
628                          dx_cols = dplyr::starts_with("dx"),
629                          pc_cols = dplyr::starts_with("pc"),
630                          icdv    = 10)}, torture = 100)
```

631
632 In Figure 5 you can see a visual depiction of memory allocation, known as a "Flame Graph", as
633 well as execution time and call stack. By clicking on each item in the stack you will be taken
634 directly to the relevant source code and can see which portions of the code take the most time or
635 memory allocations. Figure 6 is a depiction of the data view which shows just the memory
636 changes, execution time, and source file.
637

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

644

```
645    library(tictoc)
646    tic("timing: r version")
647    out <- dplyr::bind_cols(ids, ccc_mat_r(dxmat, pcmat, icdv))
648    toc()
649
650    tic("timing: c++ version")
651    dplyr::bind_cols(ids, ccc_mat_rcpp(dxmat, pcmat, icdv))
652    toc()
653
654    timing: r version: 37.089 sec elapsed
655    timing: c++ version: 5.087 sec elapsed
656
```

657   As with previous timings, while we're showing `pccc` calls, any custom function of block of code
658   you have can be compared against an alternative version to see which performs better. The above
659   blocks of code call the core functionality of the `pccc` package – one implemented all in R, the
660   other with C++ for the matrix processing and string matching components; see sourcecode
661   available at https://github.com/magic-lantern/pccc/blob/no_cpp/R/ccc.R for full listing.

662

663   After starting with high level timings, next run benchmarks on specific sections of code such as
664   in this example comparing importing a package vs using the package reference operator using
665   bench:

666

```
667    library(bench)
668    set.seed(42)
669    bench::mark(
670      package_ref <- lapply(medium_input, function(i) {
671        if(any(stringi::stri_startswith_fixed(i, 'S'),na.rm = TRUE))
672          return(1L)
673        else
674          return(0L)
675    }))
676
677    # A tibble: 1 x 14
```

```
678        expression   min  mean median   max `itr/sec` mem_alloc
679         <chr>       <bch> <bch> <bch:> <bch>     <dbl> <bch:byt>
680      1 package_r… 547ms 547ms  547ms 547ms      1.83    17.9MB
681
682      library(stringi)
683      bench::mark(
684          direct_ref <- lapply(medium_input, function(i) {
685          if(any(stri_startswith_fixed(i, 'S'),na.rm = TRUE))
686            return(1L)
687          else
688            return(0L)
689      }))
690
691      # A tibble: 1 x 14
692        expression   min  mean median   max `itr/sec` mem_alloc
693         <chr>       <bch> <bch> <bch:> <bch>     <dbl> <bch:byt>
694      1 direct_re… 271ms 274ms  274ms 277ms      3.65    17.9MB
695
```

The above test was run on a virtual machine running Ubuntu 16.04.5 LTS using R 3.4.4.

One benefit of `bench::mark` over `microbenchmark` is that bench reports memory allocations as well as timings, similar to data shown in `profvis`. Through benchmarking we found that for some systems/configurations the use of the "::" operator, as opposed to importing a package, worsened performance noticeably. Also widely known (Gillespie & Lovelace, 2017) and found to be applicable here is that the use of matrices are preferred for performance reasons over data.frames or tibbles. Matrices do have different functionality, which can require some re-work when converting from one to another. For example, a matrix can only contain 1 data type such as character or numeric; data.frames and tibbles support shortcut notations such as `mydf$colname`. 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 non-macOS Unix-based operating system (OS). 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:

718

- "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 (Kelleher & Pausch, 2005; Jones & Bonsignour, 2011).
- 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. Once you decided to put effort into optimization, make sure you test on a range of realistic configurations before deciding that an "improvement" is beneficial (Hyde, 2009).
- 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; 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

Abrahms J. 2016.Big-O notation explained by a self-taught programmer. *Available at https://justin.abrah.ms/computer-science/big-o-notation-explained.html*

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

758   Allaire JJ, Francois R, Ushey K, Vandenbrouck G, library MG (TinyThread,

759   http://tinythreadpp.bitsnbites.eu/), RStudio, library I (Intel T,

760   https://www.threadingbuildingblocks.org/), Microsoft. 2018a. *RcppParallel: Parallel*

761   *Programming Tools for "Rcpp."*

762   Allaire JJ, Xie Y, McPherson J, Luraschi J, Ushey K, Atkins A, Wickham H, Cheng J, Chang W,

763   Iannone R. 2018b. *rmarkdown: Dynamic Documents for R*.

764   Apache Software Foundation. 2018.SparkR (R on Spark) - Spark 2.3.2 Documentation.

765   *Available at https://spark.apache.org/docs/latest/sparkr.html*

766   Atchison WF, Conte SD, Hamblen JW, Hull TE, Keenan TA, Kehl WB, McCluskey EJ, Navarro

767   SO, Rheinboldt WC, Schweppe EJ, Viavant W, Young DM Jr. 1968. Curriculum 68:

768   Recommendations for Academic Programs in Computer Science: A Report of the ACM

769   Curriculum Committee on Computer Science. *Commun. ACM* 11:151–197. DOI:

770   10.1145/362929.362976.

771   Beck K, Beedle M, Bennekum A van, Cockburn A, Cunningham W, Fowler M, Grenning J,

772   Highsmith J, Hunt A, Jeffries R, Kern J, Marick B, Martin RC, Mellor S, Schwaber K,

773   Sutherland J, Thomas D. 2001.Manifesto for Agile Software Development. *Available at*

774   *http://agilemanifesto.org/*

775   Beck K, Gamma E. 1998. Test Infected: Programmers Love Writing Tests. *Java Report* 3.

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

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

778   *Futures*.

779   Bird J. 2013.Applying the 80:20 Rule in Software Development - DZone Agile. *Available at*

780   *https://dzone.com/articles/applying-8020-rule-software*

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

782   Bischl B, Lang M, Mersmann O, Rahnenführer J, Weihs C. 2015. BatchJobs and

783   BatchExperiments: Abstraction Mechanisms for Using R in Batch Environments. *Journal*

784   *of Statistical Software* 64:1–25.

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

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

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

788   Calaway R, Corporation M, Weston S. 2017. *doSNOW: Foreach Parallel Adaptor for the*
789   *"snow" Package*.

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

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

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

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

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

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

800   Dragulescu AA, Arendt C. 2018. *xlsx: Read, Write, Format Excel 2007 and Excel*
801   *97/2000/XP/2003 Files*.

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

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

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

806   Feinstein JA, Russell S, DeWitt PE, Feudtner C, Dai D, Bennett TD. 2018. R Package for
807   Pediatric Complex Chronic Condition Classification. *JAMA Pediatrics*. DOI:
808   10.1001/jamapediatrics.2018.0256.

809   Feudtner C, Christakis DA, Connell FA. 2000. Pediatric Deaths Attributable to Complex Chronic
810   Conditions: A Population-Based Study of Washington State, 1980–1997. *Pediatrics*
811   106:205–209.

812   Feudtner C, Feinstein JA, Zhong W, Hall M, Dai D. 2014. Pediatric complex chronic conditions
813   classification system version 2: updated for ICD-10 and complex medical technology
814   dependence and transplantation. *BMC Pediatrics* 14:199. DOI: 10.1186/1471-2431-14-
815   199.

816   Fucci D, Scanniello G, Romano S, Shepperd M, Sigweni B, Uyaguari F, Turhan B, Juristo N,
817   Oivo M. 2016. An External Replication on the Effects of Test-driven Development Using
818   a Multi-site Blind Analysis Approach. In: *Proceedings of the 10th ACM/IEEE*

819   *International Symposium on Empirical Software Engineering and Measurement*. ESEM

820   '16. New York, NY, USA: ACM, 3:1–3:10. DOI: 10.1145/2961111.2962592.

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

822   Gillespie C, Lovelace R. 2017. *Efficient R Programming: A Practical Guide to Smarter*

823   *Programming*. Sebastopol, CA: O'Reilly Media.

824   Glass RL. 2001. Frequently Forgotten Fundamental Facts About Software Engineering. *IEEE*

825   *Softw.* 18:112–111. DOI: 10.1109/MS.2001.922739.

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

827   Hansson DH. 2014.TDD is dead. Long live testing. (DHH). *Available at*

828   *http://david.heinemeierhansson.com/2014/tdd-is-dead-long-live-testing.html*

829   Hester J. 2018. *bench: High Precision Timing of R Expressions*.

830   Hyde R. 2009. The Fallacy of Premature Optimization. *Ubiquity* 2009:1. DOI:

831   10.1145/1569886.1513451.

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

833   *List structures.*

834   Jones C, Bonsignour O. 2011. *The Economics of Software Quality*. Addison-Wesley

835   Professional.

836   Kane M, Emerson J, Weston S. 2013. Scalable Strategies for Computing with Massive Data.

837   *Journal of Statistical Software, Articles* 55:1–19. DOI: 10.18637/jss.v055.i14.

838   Kanewala U, Bieman JM. 2014. Testing Scientific Software: A Systematic Literature Review.

839   *Information and software technology* 56:1219–1232. DOI: 10.1016/j.infsof.2014.05.006.

840   Kelleher C, Pausch R. 2005. Lowering the Barriers to Programming: A Taxonomy of

841   Programming Environments and Languages for Novice Programmers. *ACM Comput.*

842   *Surv.* 37:83–137. DOI: 10.1145/1089733.1089734.

843   Klevtsov A, Antonov A, Upravitelev P. 2018. *benchr: High Precise Measurement of R*

844   *Expressions Execution Time*.

845   Klik M, Collet Y, Facebook. 2018. *fst: Lightning Fast Serialization of Data Frames for R*.

846   Knuth DE. 1974. Structured Programming with Go to Statements. *ACM Comput. Surv.* 6:261–

847   301. DOI: 10.1145/356635.356640.

848   Koskinen J. 2015.Software Maintenance Costs. *Available at*

849   *https://wiki.uef.fi/download/attachments/38669960/SMCOSTS.pdf*

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

851   Leek JT, Peng RD. 2015. Opinion: Reproducible research can still be wrong: Adopting a

852   prevention approach. *Proceedings of the National Academy of Sciences* 112:1645–1646.

853   DOI: 10.1073/pnas.1421412111.

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

855   Luraschi J, Kuo K, Ushey K, Allaire JJ, Macedo S, RStudio, Foundation TAS. 2018. *sparklyr: R*

856   *Interface to Apache Spark.*

857   Matloff N. 2016. Software Alchemy: Turning Complex Statistical Computations into

858   Embarrassingly-Parallel Ones. *Journal of Statistical Software* 71:1–15. DOI:

859   10.18637/jss.v071.i04.

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

861   Micro Focus. 2018.Unified Functional Testing. *Available at https://software.microfocus.com/en-*

862   *us/products/unified-functional-automated-testing/overview* (accessed April 12, 2018).

863   Moilanen J. 2014.Test Driven Development details. *Available at*

864   *https://github.com/educloudalliance/educloud-development/wiki/Test-Driven-*

865   *Development-details* (accessed April 12, 2018).

866   Nolan R, Padilla-Parra S. 2017. exampletestr—An easy start to unit testing R packages.

867   *Wellcome Open Research* 2. DOI: 10.12688/wellcomeopenres.11635.2.

868   Nutter B, Lane S. 2018. *redcapAPI: Accessing data from REDCap projects using the API.* DOI:

869   10.5281/zenodo.11826.

870   Osborne JM, Bernabeu MO, Bruna M, Calderhead B, Cooper J, Dalchau N, Dunn S-J, Fletcher

871   AG, Freeman R, Groen D, Knapp B, McInerny GJ, Mirams GR, Pitt-Francis J, Sengupta

872   B, Wright DW, Yates CA, Gavaghan DJ, Emmott S, Deane C. 2014. Ten Simple Rules

873   for Effective Computational Research. *PLoS Computational Biology* 10. DOI:

874   10.1371/journal.pcbi.1003506.

875   Prins P, de Ligt J, Tarasov A, Jansen RC, Cuppen E, Bourne PE. 2015. Toward effective

876   software solutions for big biology. *Nature Biotechnology* 33:686–687. DOI:

877   10.1038/nbt.3240.

878   Python Wiki Contributors. 2018.Performance Tips. *Available at*

879   *https://wiki.python.org/moin/PythonSpeed/PerformanceTips* (accessed April 12, 2018).

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

882 Ranorex GmbH. 2018.Ranorex. *Available at https://www.ranorex.com* (accessed April 12,
883   2018).

884 Reese J. 2018.Best practices for writing unit tests. *Available at https://docs.microsoft.com/en-*
885   *us/dotnet/core/testing/unit-testing-best-practices*

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

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

891 Sandve GK, Nekrutenko A, Taylor J, Hovig E. 2013. Ten Simple Rules for Reproducible
892   Computational Research. *PLOS Computational Biology* 9:e1003285. DOI:
893   10.1371/journal.pcbi.1003285.

894 Selenium Contributors. 2018.Selenium. *Available at https://www.seleniumhq.org* (accessed April
895   12, 2018).

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

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

900 Stack Overflow Contributors. 2017.Unit testing Anti-patterns catalogue. *Available at*
901   *https://stackoverflow.com/questions/333682/unit-testing-anti-patterns-catalogue*
902   (accessed April 12, 2018).

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

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

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

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

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

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

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

911   Wickham H, RStudio, Feather developers, Google, LevelDB Authors. 2016. *feather: R Bindings*
912   *to the Feather "API."*
913   Wikipedia contributors. 2017a.SUnit — Wikipedia, The Free Encyclopedia. *Available at*
914   *https://en.wikipedia.org/w/index.php?title=SUnit&oldid=815835062*
915   Wikipedia contributors. 2017b.XUnit — Wikipedia, The Free Encyclopedia. *Available at*
916   *https://en.wikipedia.org/w/index.php?title=XUnit&oldid=807299841*
917   Wilson G. 2016. Software Carpentry: lessons learned. *F1000Research*. DOI:
918   10.12688/f1000research.3-62.v2.
919   Wilson G, Aruliah DA, Brown CT, Hong NPC, Davis M, Guy RT, Haddock SHD, Huff KD,
920   Mitchell IM, Plumbley MD, Waugh B, White EP, Wilson P. 2014. Best Practices for
921   Scientific Computing. *PLOS Biology* 12:e1001745. DOI: 10.1371/journal.pbio.1001745.
922   Xie Y. 2018. *testit: A Simple Package for Testing R Packages*.
923   Xie Y, Allaire JJ, Grolemund G. 2018. *R Markdown: The Definitive Guide*. Boca Raton, Florida:
924   Chapman and Hall/CRC.
925   Xochellis J. 2010.The impact of the Pareto principle in optimization - CodeProject. *Available at*
926   *https://www.codeproject.com/Articles/49023/The-impact-of-the-Pareto-principle-in-*
927   *optimization*
928   Yu H. 2002. Rmpi: Parallel Statistical Computing in R. *R News* 2:10–14.
929

Figure 1

Packages with non-empty testing directory

Count of packages with files in standard testing directories by year a package was last updated. Testing directory "Yes" is determined by the presence of files matching the regular expression "[Tt]est[^/]*/.+"; if no matches are found for an R package, is it counted as a "No."
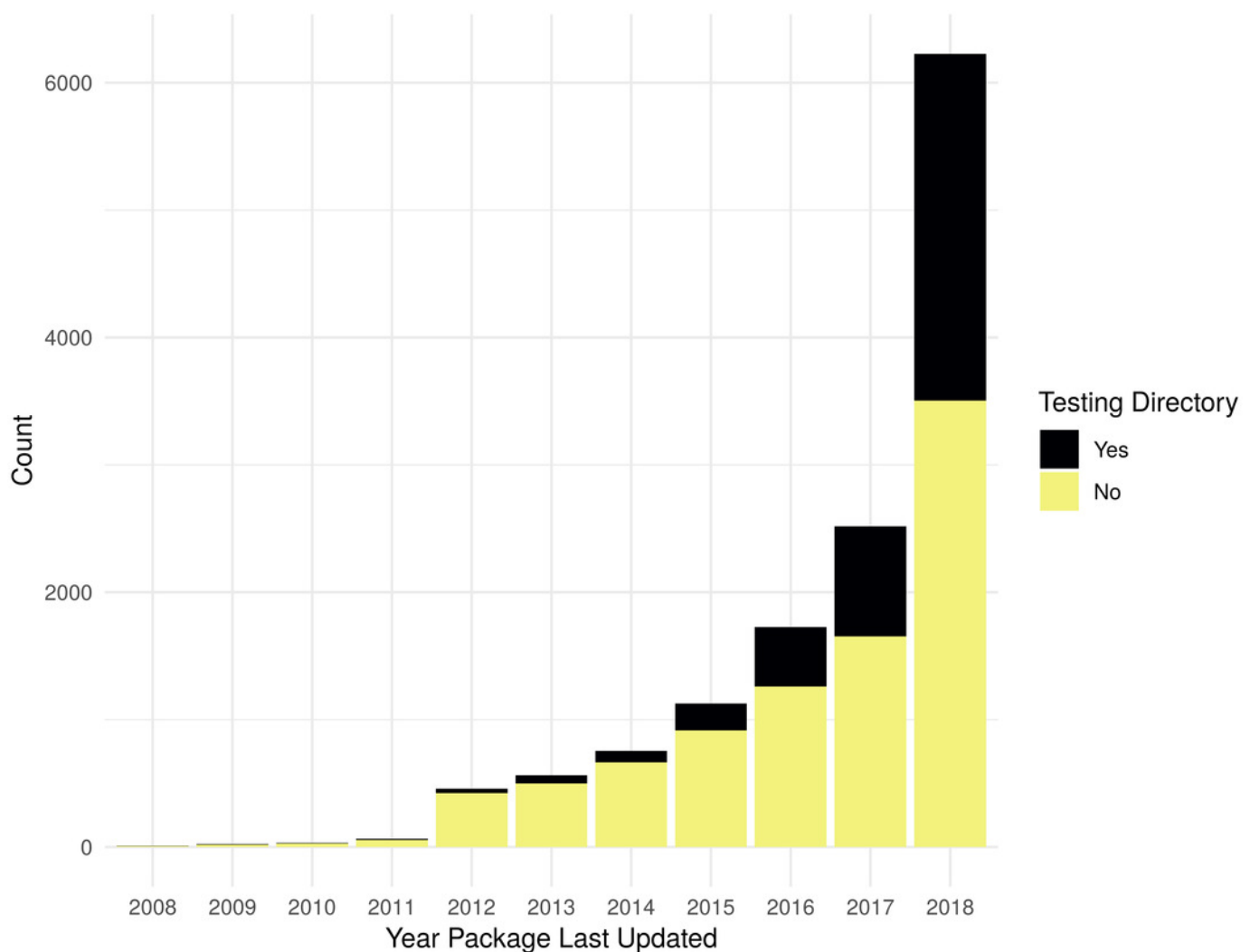
Figure 2

Packages with testing framework dependency

Count of dependencies on a testing package (RUnit, svUnit, testit, testthat, unitizer, unittest) by year a package was last updated. Packages with no stated dependency from their DESCRIPTION file for one of the specified packages are listed as 'none.'
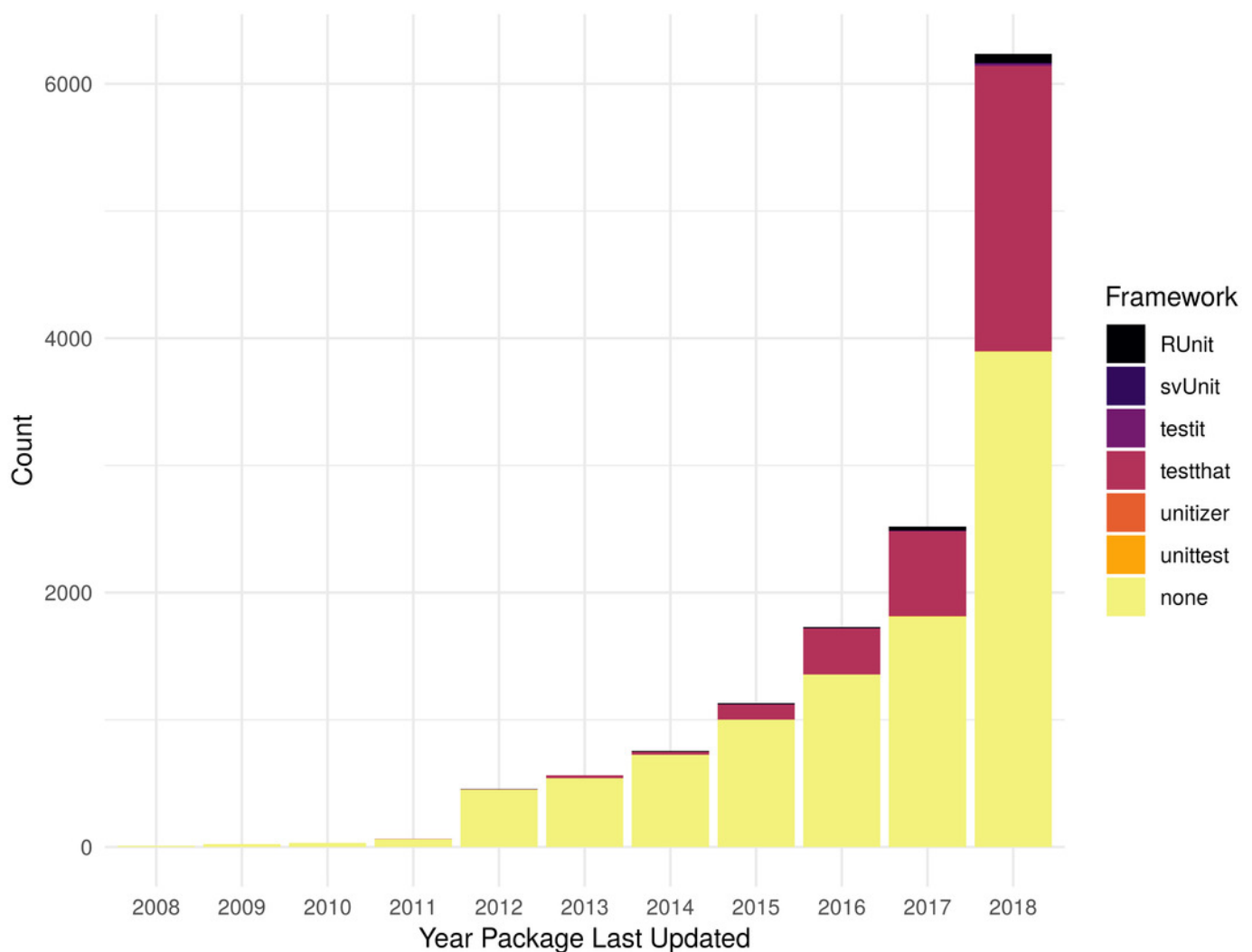
Figure 3

Packages with non-empty src directory

Figure 3 Count of packages with files in standard source directories that has code to be compiled by year a package was last updated. Compiled directory "Yes" is determined by the presence of files matching the regular expression "src[^/]*/.+"; if no matches are found for an R package, is it counted as a "No."
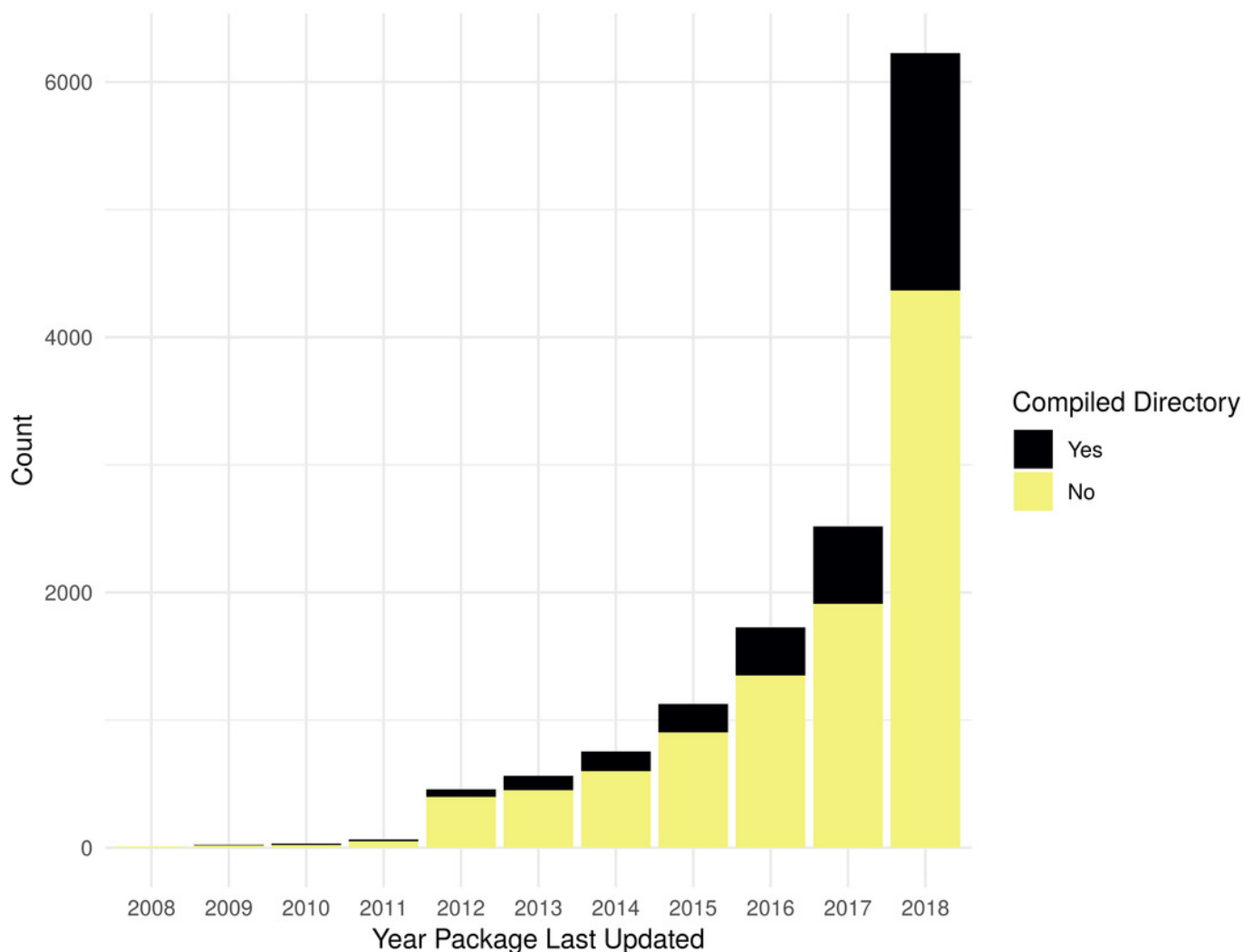
Figure 4

Packages with optimization framework dependency

Count of dependencies on an optimization related package, see "OPTIMIZATION OF R PACKAGES" section for complete list, by year a package was last updated. Packages with no stated dependency from their DESCRIPTION file for one of the specified packages are listed as 'none.' In order to aid visual understanding of top dependencies, we limited display to those packages that had 14 or more dependent packages.
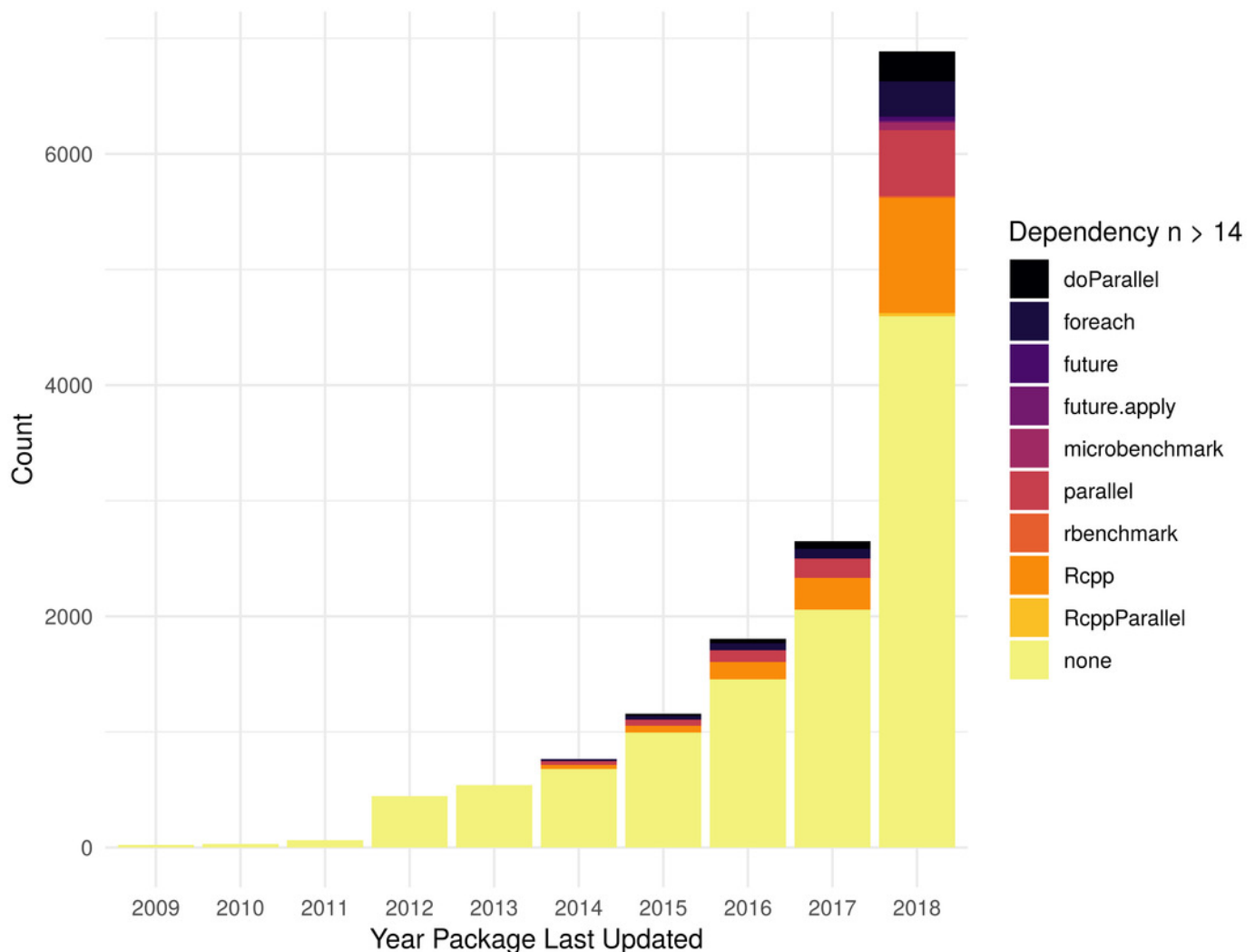
Figure 5

Profvis Flame Graph

Visual depiction of memory allocation/deallocation, execution time, and call stack.
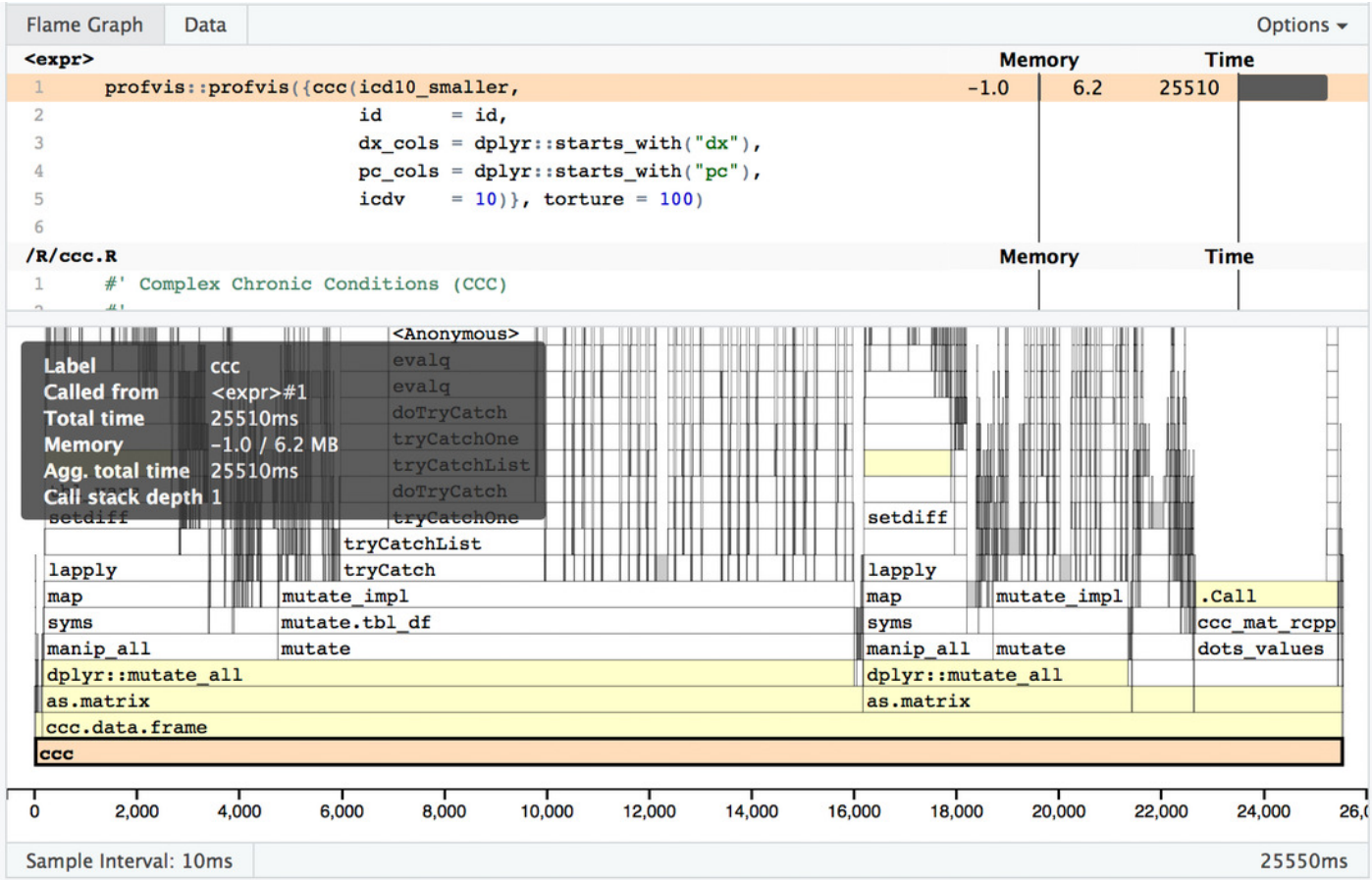
Figure 6

Profvis Data Chart

Table view of memory allocation/deallocation, execution time, and call stack.

Code	File	Memory (MB)		Time (ms)	
▼ ccc	<expr>	−1.0 ❘	6.2	25510	▪
▼ ccc.data.frame	ccc.R	−1.0 ❘	6.1	25380	▪
▼ as.matrix	ccc.R	−1.0 ❘	4.5	21270	▪
▼ dplyr::mutate_all	ccc.R	−0.9 ❘	1.1	21000	▪
▼ mutate		−0.4 ❘	0.5	13890	▪
▼ mutate.tbl_df		−0.4 ❘	0.5	13890	▪
▶ mutate_impl		−0.4 ❘	0.5	13870	▪
lazyLoadDBfetch		0 ❘	0.0	10	
▶ named_quos		0 ❘	0.0	10	
▶ manip_all		−0.5 ❘	0.6	7100	❘
lazyLoadDBfetch		0 ❘	0.0	10	
▶ as.matrix.data.frame		0.0 ❘	1.2	190	
▶ get		0.0 ❘	0.1	50	
dxmat <− as.matrix(dplyr::mutate_all(dplyr::select(data, !!dplyr::enquo(d	ccc.R	0 ❘	0.0	20	

Figure 7

Profvis Flame Graph .Call()

Visual depiction of memory allocation/deallocation, execution time, and call stack; note the limitations in detail at the .Call() function where custom compiled code is called.