The State of Software for Evolutionary Biology

Diego Darriba,[1] Tomáš Flouri,[1] and Alexandros Stamatakis*,[1,2]

[1]Scientific Computing Group, Heidelberg Institute for Theoretical Studies, Heidelberg, Germany

[2]Institute of Theoretical Informatics, Karlsruhe Institute of Technology, Karlsruhe, Germany

***Corresponding author:** E-mail: alexandros.stamatakis@h-its.org.

Associate editor: Keith Crandall

Abstract

With Next Generation Sequencing data being routinely used, evolutionary biology is transforming into a computational science. Thus, researchers have to rely on a growing number of increasingly complex software. All widely used core tools in the field have grown considerably, in terms of the number of features as well as lines of code and consequently, also with respect to software complexity. A topic that has received little attention is the software engineering quality of widely used core analysis tools. Software developers appear to rarely assess the quality of their code, and this can have potential negative consequences for end-users. To this end, we assessed the code quality of 16 highly cited and compute-intensive tools mainly written in C/C++ (e.g., MrBayes, MAFFT, SweepFinder, etc.) and JAVA (BEAST) from the broader area of evolutionary biology that are being routinely used in current data analysis pipelines. Because, the software engineering quality of the tools we analyzed is rather unsatisfying, we provide a list of best practices for improving the quality of existing tools and list techniques that can be deployed for developing reliable, high quality scientific software from scratch. Finally, we also discuss journal as well as science policy and, more importantly, funding issues that need to be addressed for improving software engineering quality as well as ensuring support for developing new and maintaining existing software. Our intention is to raise the awareness of the community regarding software engineering quality issues and to emphasize the substantial lack of funding for scientific software development.

Key words: **software engineering quality, scientific computing, data analysis, numerical analysis, policy issues, evolutionary inference software.**

Introduction

With Next Generation Sequencing data (NGS) coming off age and being routinely used, it cannot be disputed that evolutionary biology is becoming an increasingly quantitative and computational discipline (see Barone et al. 2017). Thus, quantitative as well as computational skills are increasingly foundational for the discipline.

The field is also transforming into a computational science as it increasingly relies on cluster computers and supercomputers (e.g., Misof et al. 2014 or Jarvis et al. 2014). This is a transition other disciplines such as astrophysics or fluid dynamics underwent decades ago. Our perception is that there exists a lack of funding for accomplishing this transition.

The common denominator of the above trends is that researchers have to rely on a growing number of increasingly complex core software components. By software complexity we refer to the fact that all widely used tools have grown considerably, in terms of the number of features as well as lines of code (LoC). For instance, MrBayes (Ronquist et al. 2012) had ~49,000 LoC in 2005 and ~94,000 in 2014. Phylogenetic inference software now supports a substantially larger set of models (e.g., substitution models), hardware platforms (e.g., GPUs, clusters, etc.), and types of parallelism (e.g., fine-grain, coarse-grain, hybrid approaches).

In addition, software complexity can also be quantified by means of the core component count in current analysis pipelines. For instance, in the "Sanger days," the analysis pipeline was rather straightforward, once the sequences were available. For a phylogenetic study it consisted of the following steps: align → infer tree → visualize tree. For NGS data and huge phylogenomic data sets, such as the insect transcriptome (Misof et al. 2014) or bird genome evolution (Jarvis et al. 2014) projects, the data analysis pipelines have become substantially longer and more complex. They also require user expertise in an increasing number of Bioinformatics areas (e.g., orthology assignment, read assembly, data set assembly, partitioning of data sets, divergence time estimates, etc.). In addition, these pipelines require a plethora of helper scripts to transform formats, partially automate the workflows, and to connect the components. Helper scripts are typically written in languages such as perl, a language that is highly susceptible to coding errors due to lack of typing, or python that uses dynamic typing and can thus not be subjected to a comprehensive type-check either. The term "typing" refers to the data types of variables (e.g., integer or floating point) that are passed to and returned by functions. Without strict typing, a function expecting an integer argument can be invoked with a floating point value as an argument and exhibit undefined or unexpected behavior. Thus, programming

Review

languages with stricter type control can reduce the potential for errors.

Our main concern is that, if each code (henceforth, we use code as synonym for software) or script component *i* used in such a pipeline has a probability of being "buggy" P_i, the probability that there is a bug in the pipeline increases dramatically with the number of components. Here, we refer to bugs that do not lead to crashes which are easily detected and fixed, but to bugs that yield incorrect results. In ExaML (Kozlov et al. 2015), for instance, we detected two major bugs in branch length scaling and bootstrap replicate generation under specific settings while conducting large-scale inferences on empirical data. Thus, if detected too late, errors, and particularly those in early pipeline stages (e.g., NGS assembly) for large-scale data analysis projects can have a dramatic impact on downstream analyses such as phylogenetic inferences or dating as they will all have to be repeated. Wilson et al. (2017) provide valuable general recommendations on good practices in scientific computing and for designing as well as managing scientific workflows. Given that our field needs to compete with established computational sciences for scarce supercomputing or cloud resources, repeating large phylogenomic analyses can result in a substantial waste of computational effort. Current large-scale phylogenomic analysis projects can require between 10 and 70 million processor hours on supercomputers.

Our goals in this paper are to (1) assess the software engineering quality of current tools irrespective of their accuracy or algorithmic quality and (2) to propose potential solutions, including software analysis tools, for improving the quality of evolutionary biology software. We wish to emphasize that the quality measures we deploy only represent one possible option for assessing software engineering quality. Also, poor software engineering quality does not automatically induce that software is incorrect, yet a significant link *does* exist (e.g., Briand et al. 1999, 2000; Casalnuovo et al. 2015). Note that, our criteria for software engineering quality differ from the aforementioned papers. They have, however, in part been motivated by these.

There is little related work on the topic of software engineering quality in Bioinformatics. Kumar and Dudley (2007) discuss Bioinformatics software engineering quality in terms of usability and technical requirements from the perspective of the end user. Rother et al. (2012) describe a software engineering toolbox for developing Bioinformatics software that reviews development practices in the author's own codes and projects. While situated at a higher level of abstraction than our work, the strategies proposed in this paper can be useful for planning and managing new software projects. Leprevost et al. (2014) also discuss some best practices for Bioinformatics software development, but in a rather general setting, without providing specific suggestions on how to improve quality. Wilson et al. (2014) provide a list of generic best practices for scientific software development and list examples of several high-profile paper retractions due to erroneous software.

With respect to software verification and testing, Giannoulatou et al. (2014) describe the application of the so-called metamorphic testing approach (Chen et al. 1998) to the BWA, Bowtie, and Bowtie2 short read mappers. See also Chen et al. (2009) for another application of metamorphic testing to Bioinformatics software. Metamorphic testing can only be applied either to a single tool, or to a set of tools that serve the same purpose. Finally, Kamali et al. (2015) provide an overview of different testing techniques and discuss their applicability to Bioinformatics software. We deliberately do not focus on assessing result quality as this substantially limits the scope of tools we can assess (e.g., only phylogenetics tools) and the majority of the tools we selected is well-tested. However, tests exclusively relying on simulated data need to be treated with caution as there is no guarantee that data simulation tools are implemented correctly.

For assessing software engineering quality we downloaded and scrutinized—using a common set of criteria—16 frequently used and cited codes that often form the basis of data analyses in evolutionary biology. With the exception of BEAST (written in JAVA) and despite the emergence of languages such as R we focus on software written in C/C++ as this is the predominant programming language of the highly popular and compute-intensive tools we tested. We mainly focus on compute-intensive tools because errors in these tools imply a substantial waste of computational resources (e.g., CPU time grants) if analyses need to be repeated. For comparison, we also analyzed an Astrophysics code developed at our research institute because Astrophysics is a more mature computational science discipline. On the basis of the software analysis results, we provide our personal and subjective list of best practices and discuss some science policy issues that need to be addressed for improving software engineering quality and for better supporting scientific software development.

It is absolutely not our intention to criticize any of the authors and developers of the codes we assessed as they have all made major contributions to the field. It is quite typical that the careers of principal investigators in Bioinformatics are based on one or more widely used tools they have developed. As they become more senior and manage larger research groups, there is less time available to maintain and occasionally redesign the tools, despite the fact that they know how to properly write software. In addition, they are mostly reluctant to delegate this task to graduate students or postdoctoral researchers because they should work on more interesting projects instead of merely re-engineering widely used software.

Given that most software for evolutionary biology is distributed under the GNU GPL license, users and critics should keep the following quote from the GNU GPL license in mind: "The copyright holders and/or other parties provide the program 'as is' without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the program is with you. Should the program prove defective, you assume the cost of all necessary servicing, repair, or correction."

Thus, our goal is to emphasize that users should be aware of the fact that software is imperfect and take software

Table 1. Evaluated Software Packages per Application Domain Including Version Numbers.

Domain	Software	Version Number
Phylogenetics	PAML (Yang 2007)	4.8
	PHYML (Guindon et al. 2010)	20141009
	MrBayes (Ronquist et al. 2012)	3.2.4-svn(r926)
	RAxML (Stamatakis 2014)	8.2.11
Population genetics	MS (Hudson 2002)	Sep 8, 2014
	SweepFinder (Nielsen et al. 2005)	Feb 23, 2015
Seq. alignment	MAFFT (Katoh and Standley 2013)	7.205
	T-Coffee (Notredame et al. 2000)	20141026_23: 18
	Prank (Löytynoja and Goldman 2005)	140603
Div. times	Beast (Drummond and Rambaut 2007)	1.8.0
	FDPPDIV (Heath et al. 2014)	1.3
Multi.-Sp. coalescence	BP&P (Yang and Rannala 2010)	3.0
Seq. simulation	Seq-Gen (Rambaut and Grass 1997)	1.3.3
	INDELible (Fletcher and Yang 2009)	1.0.3
De novo assembly	SOAP (Li et al. 2009)	r240
	Abyss (Simpson et al. 2009)	1.5.2
Astrophysics	Gadget-2 (Springel 2005)	2.0.7

Note.—As MS and SweepFinder do not have version numbers we show the download dates.

engineering quality into account when selecting tools for data analyses. Furthermore, because of the increasing reliance on software in current day biology, there exists a substantial funding, sustainability, and maintenance issue that needs to be addressed.

Software and Analysis Methods

Software

We selected highly cited open-source tools from the following areas: phylogenetic inference, population genetics, multiple sequence alignment, divergence time estimation, multi-species coalescence, sequence simulation, and de novo assembly. Note that tools from all of these areas can be used in evolutionary biology data analysis pipelines. We did not modify our RAxML source code such that it compares favorably to the other tools.

Table 1 lists the codes we assessed in each domain.

Code Analysis Criteria

Since we analyzed a comparatively large number of codes, we had to deploy rather simple and straightforward techniques to analyze them. This approach is generally known as the empirical software engineering and metrics approach. There exist dedicated conferences on this topic in the broader area of software engineering (e.g., http://www.esem-conferences.org/; last accessed February 11, 2018). Researchers in this field also increasingly browse open-source code on github to obtain quantitative data about software. Alternatively, we could have analyzed one or two tools in greater depth, but our goal was to obtain a general overview of software engineering quality in the field.

Initially, we compiled all codes using the standard GNU compilers (gcc/g++) as well as the clang compiler by Apple. We enabled all reasonable warning flags in the two C/C++ compilers as well as analogous flags in JAVA for analyzing BEAST (see supplementary material, Supplementary Material online for details). We subjectively classified GNU compiler warnings into major warnings that are potentially dangerous and minor warnings that are less dangerous, but should be fixed nonetheless (see supplementary material, Supplementary Material online for our classification criteria). We count and classify compiler warnings, because we assume that the more warnings a code produces, the more likely it is to behave in an unexpected way. However, this does not automatically induce that the results computed by these codes are incorrect, since a code that produces no warnings can yield incorrect results.

In addition, we executed the codes using valgrind (http://valgrind.org/; last accessed February 11, 2018), a tool that detects potential memory leaks, illegal memory accesses, lost memory blocks, etc. We classified results into three categories: "clean" when running the codes with valgrind did not generate any warnings, "invalid" for read or write accesses at an invalid RAM (Random Access Memory) address, or "leaks" when allocated memory was not properly freed again. Memory errors or incorrect usage of memory serves as an indicator for the probability of crashes or unspecified/undefined behavior, when accessing values at invalid or uninitialized RAM locations.

Thereafter, we used the grep text searching tool to identify a typical programming error associated with the C malloc() routine that is used to allocate a memory block of *n* bytes in RAM. Frequently, this function is invoked with integer data types that are too small for representing *n* to allocate large chunks of memory. In our analyses, we distinguish between three malloc() usage errors: "NoCast" (i.e., missing typecast) and "MisCast" (misplaced cast) and "WrongCast" (incorrect cast). For the new[] operator in C++ we use an analogous classification. Examples for these error types (e.g., in MrBayes and ms) are provided in the supplementary material, Supplementary Material online. While for smaller data sets this incorrect usage will have no effect, programs are likely to fail when deployed for analyzing NGS data sets on powerful multi-core servers which are nowadays often equipped with 128 or 256GB RAM. In fact, we have experienced such crashes

Table 2. PAML Components.

PAML component	LoC (own)	LoC (total)	Major W.	Minor W.	Clang W.	Malloc	Valgrind	Assert
baseml	1,304	14,212	0.0	4.6	623.0	NoCast	Clean	0.0
basemlg	685	13,593	7.2	4.37	452.7	NoCast	Leaks	0.0
chi2	185	185	0.0	27.0	37.85	NoCast	Clean	0.0
codeml	5,309	18,217	4.7	8.5	229.62	NoCast	Clean	0.0
evolver	1,123	14,031	4.5	58.8	297.6	NoCast	Leaks	0.0
mcmctree	2,970	8,079	2.4	11.1	184.1	NoCast	Clean	0.0
pamp	514	13,422	1.9	9.7	485.4	NoCast	Leaks	0.0
yn00	712	927	4.2	50.8	315.5	NoCast	Leaks	0.0

NOTE.—LoC(own) is the number of effective lines of code that belong only to the component. LoC(total) is the total number of effective lines of code for each component, including code shared with other components. Columns "Major W." and "Minor W." give the major and minor GNU compiler warnings and "Clang W." reports the number of clang warnings, all normalized to 1,000 lines of own code. Column "Malloc" provides the malloc() casting error, "Valgrind" the memory behavior and "assert" the number of assertions per 1,000 lines of code.

with our own ExaML (Kozlov et al. 2015) code as well as with MrBayes.

Another code feature that we consider as being important is the use of so-called assertions (e.g., the assert() function in C, see supplementary material, Supplementary Material online for a classic assert() example). We assessed the usage of assertions by calculating the number of assertions per 1,000 LoC. Assertions contain logical clauses about variables that must be true when the program conducts an assertion call, otherwise the program fails. The use of assertions is associated with code correctness. In theoretical computer science, there exists a formal framework, the so-called Hoare logic (Hoare 1969), for proving program correctness. It works by inserting assertions (Boolean statements about variable states) at appropriate positions in the code and proving that they will never fail. While proving the correctness of the complex scientific codes we scrutinize here using Hoare logic is not feasible, the frequent use of assertions in a program *is* an indicator of code quality. Assertions are also helpful for documenting and debugging code as the part that fails can easily be identified. A recent software engineering study using a large collection of C/C++ codes from github suggests that functions *with* assertions *do* have significantly fewer defects in collaborative development projects (Casalnuovo et al. 2015) which are also common in Bioinformatics.

To obtain a rough estimate of code complexity, we also counted the LoC in each of the programs using the cloc (https://github.com/AlDanial/cloc; last accessed February 11, 2018) command that excludes comments and empty lines. For some programs we also generated histograms that illustrate code growth over the last years (see supplementary material, Supplementary Material online). The LoC metric does, of course, not directly reflect code complexity, but can serve as a rough proxy.

A more elaborate criterion for assessing code complexity is the degree of code duplication, that is, how many copies of identical code are present in the source files. In general, code duplication represents a bad programming practice. If a bug is detected and fixed in one copy of the duplicated code, it needs to be fixed in all duplicates. Mostly, these duplicates are not properly documented and potentially difficult to find. Thus, software with a high degree of code duplication is more difficult to maintain and thus also more likely to contain errors. For instance, a large-scale software engineering study (Juergens et al. 2009) of commercial and open source software found that (1) inconsistent changes to code clones are very frequent and (2) induce a significant number of faults.

Overall, the above criteria have been selected (1) because they are easy to apply to a large number of diverse codes and because (2) there exists a link (e.g., Briand et al. 1999, 2000; Casalnuovo et al. 2015; Juergens et al. 2009) between quality and the probability of erroneous program behavior, that is, crashes or calculation of incorrect results.

There also exist more elaborate methods for analyzing and improving code quality such as the pmccabe tool, for instance, that assesses function complexity.

Software Analysis Results

A detailed analysis of all codes, including source code examples where appropriate, is provided in the supplementary material, Supplementary Material online.

We summarize the results from our standard tests in table 2 for individual PAML components and in table 3 for all other programs including the PAML core code. The results obtained by the Simian tool (http://www.harukizaemon.com/simian/; last accessed February 11, 2018) that reports the degree of code duplication are summarized in table 4.

One general observation is that the clang compiler issues substantially more warnings than the GNU compilers. This is because it performs a more thorough static code analysis than gcc, that is, a more in depth check, including stricter type checking. Another general trend is the infrequent use of assertions as well as a rather sloppy memory management. While memory leaks can be harmless, invalid memory accesses (Prank, MrBayes, MAFFT) are likely to yield unspecified behavior. In fact, most bugs in C code used to be memory-related (Lu et al. 2005) but there also seems to be a tendency for them to decrease due to the availability of tools such as valgrind (Li et al. 2006) which are apparently not used on a regular basis for developing the majority of the tools we have tested here.

We also observe a high degree of code duplication in some codes (e.g., MrBayes, SOAP, MAFFT, Prank, BEAST).

Overall, the perfect software does not seem to exist, with the exception of Abyss maybe, if we ignore the clang warnings. The Astrophysics code is not perfect either (e.g., using no

Table 3. PAML Values Refer to Parts of the Source Code that is Shared Among All Individual Components that are Listed in table 2.

Code	Language	LoC	Major W.	Minor W.	Clang W.	Malloc	Valgrind	Assert
PAML	C	12,908	0.9	9.4	18.8	NoCast	clean	0.0
PHYML	C	56,456	0.0	0.0	56.5	NoCast	clean	0.16
MrBayes	C	94,432	0.02	0.0	9.6	MisCast	Invalid/leaks	2.37
RAxML	C	57,233	0.0	0.0	16.8	No-Error	Leaks	17.5
SOAP	C/C++	37,020	3.9	17.0	155.5	NoCast	Leaks	0.0
Abyss	C	43,189	0.0	0.0	134.8	No-Error	Clean	23.11
MS	C	2,063	4.8	10.7	62.3	WrongCast	Leaks	0.0
SweepFinder	C	4,465	0.0	32.3	52.4	NoCast	Clean	1.56
MAFFT	C	57,688	1.1	1.3	27.3	NoCast	Invalid/leaks	0.0
T-Coffee	C	160,223	2.2	3.9	34.2	NoCast	Leaks	0.44
Prank	C++	23,947	6.8	0.3	121.4	NoCast	Invalid	9.19
BEAST	JAVA	302,611	0.07	12.5	N/A	No-Error	N/A	0.0
FDPPDIV	C++	11,474	3.0	3.5	61.7	No-Error	Leaks	0.26
BP&P	C	16,593	3.0	5.8	49.0	NoCast	Leaks	0.0
Seq-Gen	C	3,977	0.0	1.0	51.3	No-Error	Leaks	0.0
INDELible	C++	11,402	0.0	22.8	182.5	No-Error	Clean	0.0
Gadget-2	C	12,509	0.0	2.9	48.8	NoCast	Probably clean	0.0

NOTE.—Column "Language" denotes the programming language and column "LoC" is the total number of effective lines of code. Columns "Major W." and "Minor W." give the major and minor GNU compiler warnings and "Clang W." reports the number of clang warnings, all normalized to 1,000 lines of code. Column "Malloc" provides the malloc() casting error, "Valgrind" the memory behavior. We denote the Gadget-2 code as "probably clean" since we interrupted the valgrind analysis that did not report any errors after 30 min of run-time. Finally, column "assert" represents the number of assertions per 1,000 lines of code.

assertions at all), despite the fact that it comes from a more traditional field of computational science. We believe that our set of criteria allows to identify potential problems that can, in most cases, easily be fixed.

Discussion

We have scrutinized 16 widely used codes for evolutionary data analyses using a simple set of tools and criteria that can be deployed to improve code quality, sometimes without fully understanding the source code. Evidently, software engineering quality can only be assessed with open-source codes, hence we strongly advocate in favor of open-source such that users do have a chance to assess code quality.

We have detected several errors that are common to almost all tools and that are comparatively easy to fix. Again, we do not intend to criticize the authors of the tools, given their time and resource constraints with respect to extending and maintaining software. We want to emphasize that more awareness about code quality and, perhaps more importantly, worrying about correctness is necessary (albeit the tools we scrutinized *do* yield high quality results) since the research produced by our community heavily relies on the results produced by an entire zoo of core tools in long analysis pipelines.

Another concern is that evolutionary analysis software is frequently used as a black box with default parameters and without a proper understanding of the underlying theory or algorithms. Given the large set of tools modern evolutionary biologists need to deploy to "get a paper published," this user behavior is nonetheless understandable. There is an evident trade-off between the thoroughness of computational analyses and the publication rate. While this is difficult to change, the issue could be addressed at the teaching level. Our perception is that graduate and undergraduate training in biology needs to focus more on covering mathematical and computational topics.

We initially discuss some good practices for code development in the hope that they will be broadly adopted by the community and that they might help to reduce the number of potential bugs. Then, we discuss issues pertaining to floating-point arithmetics and reproducibility of numerical results. Finally, we discuss funding policy issues, that is, what sort of mechanisms might be required to ensure sustainable maintenance, support, and quality improvements in scientific software.

Basic Best Practices

Readers should keep in mind that our recommendations are subjective as they are based on our proper programming experience and on empirical software engineering studies. Some recommendations can be directly derived from the simple criteria we have deployed. Therefore, a good code should:

- be compiled with all compiler warning flags enabled using several compilers (e.g., icc, clang, gcc)
- should be analyzed with valgrind for memory leaks and invalid read/write accesses
- should be checked for malloc() type casting errors
- should use as many assertions as necessary and feasible to ensure the correctness of the algorithm

It might represent a good idea to ask reviewers of Bioinformatics software papers to check software they review according to the above straightforward criteria. Alternatively, journals could impose upon authors that the codes they submit for publication need to be compiled and checked accordingly prior to submission. This could be implemented by asking authors to provide appropriate code quality analysis transcripts. Part of these processes could also be automated. Finally, one should also put special emphasis on software engineering quality issues (e.g., no clang warnings, usage of

Table 4. Results of a Code Duplication Analysis Using the Simian Tool.

Code	Lines Checked	Files Checked	Duplicate Lines	Duplication %	Blocks	Files
PAML	22,200	17	1,210	5.5%	120	11
PHYML	42,786	73	5,878	13.7%	549	32
MrBayes	70,680	19	21,862	30.9%	1,680	10
RAxML	55,873	25	17,137	30.7%	1,304	22
SOAP	27,514	116	10,107	36.7%	527	72
Abyss	37,038	212	4,245	11.5%	441	71
MS	1,718	24	186	10.8%	21	9
SweepFinder	3,777	12	293	7.8%	28	3
MAFFT	45,045	72	28,630	63.6%	1,647	59
T-Coffee	82,758	196	19,345	23.4%	1,325	58
Prank	16,124	67	5,318	33.0%	462	43
BEAST	228,316	2,336	64,024	28.0%	4,786	1,151
BP&P	14,332	5	502	3.5%	56	3
Seq-Gen	3,244	44	206	6.4%	25	6
INDELible	9,840	7	1,954	19.9%	106	5
Gadget-2	9,770	31	3,314	33.9%	180	31

NOTE.—The column "Lines checked" refers to the total number of source lines and "Files checked" to the total number of source files analyzed with Simian. Note that, the "Lines checked" number is not identical to the LoC numbers reported in tables 2 and 3, since the Simian tool does not take header files into account. Column "Duplicate lines" provides the number of duplicate lines detected, "duplication %" the relative amount of code duplication, and "Blocks" provides the total number of contiguous duplicated blocks of code. Finally, column "Files" gives the number of files in which duplicated code was detected.

assertions, checks with valgrind) when teaching programming practicals at the graduate and undergraduate level.

Assertions are also particularly useful for debugging, since users often provide incomplete bug reports. In contrast to this, when an assertion fails, users will typically report the failed assertion including the source file name and the line in the code which substantially accelerates problem identification. Also, assertions are the only mechanism we considered that helps to partially assess actual code correctness and not only identify potential programming errors or bad programming practice.

While invalid read/write accesses need to be fixed, memory leaks, in particular when programs do not free all the memory they use upon termination (e.g., several PAML components and RAxML), should be addressed as well. Such program termination leaks may become problematic when one intends to integrate leaky code as a library component into some larger project. Unfortunately, it is always hard to predict which software one writes will become widely used and how much effort should be spent on code quality.

The above best practices can be easily applied without investing too much effort but will certainly improve code quality as well as help to reduce the number of implementation-induced bugs. Evidently, we also need to worry about conceptual errors that affect correctness, such as the (for a long time undetected) error in Hastings ratio calculations (Holder et al. 2005) in Bayesian inference programs.

Advanced Best Practices

Another question is what else *could* be done to improve code quality in an ideal setting. Users often tend to forget that many codes, specifically in population genetics and phylogenetics, use statistical models defined on real numbers. As a consequence, they are at the mercy of floating point arithmetics with round-off errors and numerical under or overflows. Therefore, every programmer in the field should read the classic paper "What Every Computer Scientist Should Know About Floating Point Arithmetic" by Goldberg (1991). The most important property one should be aware of is that in floating point arithmetics associativity (i.e., $(x + (y + z)) = ((x + y) + z)$) does not necessarily hold because of round-off errors. Note that, the order of arithmetic operations and thus the degree of deviations due to round-off errors depends on (1) the compiler used (2) the hardware features that are being used, and (3) on how the programmer orders the arithmetic operations. Therefore, different ML program implementations (e.g., RAxML and PHYML) can yield different log likelihood scores.

However, even the same program can return different values when the likelihood calculations are parallelized over sites, depending on the number of processors being used due to round-off errors. Thus, different numbers of processors can yield different tree topologies and, as a consequence, ML inference results may not be reproducible, even if (1) *exactly* the same tree search heuristic is applied and (2) likelihood calculations are generally sufficiently accurate despite round off errors. For instance, we executed the AVX version of RAxML twice (data available at https://github.com/stamatak/softwareQuality; last accessed February 11, 2018), once in the sequential version and once with the PThreads version as follows:

```
raxmlHPC-AVX -p 12345 -m GTRGAMMA
-s 354 -n T1
raxmlHPC-PTHREADS-AVX -T 2 -p 12345
-m GTRGAMMA -s 354 -n T2
```

The only difference between the two calls is that the addition order of per-site log likelihoods and per-site derivatives for optimizing branch lengths is changed due to the parallelization. The data set we used is a single-gene alignment of 354 *ITS* sequences with 460 sites (Grimm et al. 2006) that was known to have a "rough" likelihood surface. In other words, it

exhibits numerous local maxima that cannot be distinguished from each other using statistical significance tests. Simply because the numerical deviations make the tree searches follow distinct paths, the two, in theory identical invocations, yield different final trees with log likelihood scores of −6562.158295 versus −6562.158171 and a relative Robinson–Foulds distance (Robinson and Foulds 1981) of 8.26%. Of course, any likelihood-based significance test comparing the two trees shows that they are not significantly different from each other.

As a consequence, in an ideal world we should also carry out a theoretical round-off error analysis for our codes. As shown above, this is particularly critical for ML codes that strive to obtain a single point estimate. Round-off error analyses have been conducted decades ago for classic numerical problems such as the Gram–Schmidt orthogonalization method (Abdelmalek 1971) or, more recently, for Cholesky's QR decomposition algorithm (Yamamoto et al. 2015). Numerical issues are far less problematic for Bayesian inferences because they sample a posterior probability distribution and should thus tend to also marginalize over round-off errors. In the supplementary material, Supplementary Material online we also provide an example of how so-called denormalized floating point values can affect program performance.

Finally, since the issue of software engineering quality is just emerging, it might be extremely helpful to consult with software engineering experts to discuss appropriate development models (e.g., extreme, agile or classic waterfall models) and to improve the organization of academic programmer teams with a high fluctuation (see Rother et al. 2012). In addition, there already exists a plethora of tools that can assess the quality of the given software architecture and more advanced tools for explicitly finding bugs.

For instance, there is the pmccabe tool for assessing function complexity in C and C++ codes (https://people.debian.org/~bame/pmccabe/; last accessed February 11, 2018). It calculates the so-called McCabe cyclomatic complexity (McCabe 1976) of functions. Typically, when the complexity of a function exceeds a score of 10 or 15 the function should be split into several submodules. A quick analysis of the main RAxML source file axml.c with the following command pmccabe -f axml.c revealed that in this source file alone there are 22 functions with a cyclomatic complexity score that exceeds 15.

Furthermore, static code analysis tools analogous to the seminal Lint (Johnson 1977) tool should be deployed. The clang compiler partially does this, as do some Linux kernel development tools such as sparse and smatch. The coccinelle framework (Lawall et al. 2010) for assessing equivalence of code transformations might also be particularly useful. As described in the supplementary material, Supplementary Material online, FindBugs (http://findbugs.sourceforge.net; last accessed February 11, 2018) can be used for scrutinizing Java codes such as BEAST. Code duplication identification tools such as Simian should also be routinely used during code development. Finally, we recommend use of dead code identification tools that identify code that will never be executed (e.g., using the –coverage switch in gcc).

Another major method for improving code quality and being more confident about correctness is testing, such as unit tests or integration tests. There is a vast amount of research on, and methods for, software testing. A good starting point is the book on the art of software testing by Myers et al. (2011). The current testing practice in our field appears to be that testing is mostly delegated to users. However, we strongly advocate in favor of increased use of testing techniques.

Finally, we suggest to deploy multiple compilers and compiler versions to compile and test software (execute unit tests). This strategy can be seamlessly adopted by using Continuous Integration (CI) tools, such as Travis CI (https://travis-ci.org/; last accessed February 11, 2018). For instance, via this strategy we detected a bug in our own code for phylogenetic inference which caused a segmentation fault only when compiled with an older version of clang. As different compiler versions may produce binaries with operations that are ordered slightly differently, the likelihood of detecting errors increases.

Thus, for programmers, we further recommend the following best practices:

- read "What Every Computer Scientist Should Know About Floating Point Arithmetic"
- conduct a theoretical round-off error analysis
- be aware of denormalized floating point numbers and their impact on performance
- be aware of nonreproducibility of results when running parallel codes with different core counts
- talk to your local software engineering colleagues
- use static analyzers
- use dead code identification tools
- use a tool such as pmccabe iteratively during code development to keep module complexity low
- use a tool such as Simian to identify duplicated code
- use a tool such as Pylint (http://www.pylint.org/; last accessed February 11, 2018) for improving Python scripts
- systematically test software
- compare your implementation with other independent implementations
- use different compilers and compiler versions to compile and test software

Ideal Practices

Finally, if we intend to go even one step further, we can consider how software for critical systems such as aircraft autopilots is designed. Typically, a specification is provided to two or three completely independent software development teams. Then, they all develop software that complies with these specifications using different programming languages. Thereafter, given a broad range of input parameters, the outputs of all three independent implementations are compared. This ensures, with high probability, that the autopilot complies with the specification. One must keep in mind though that the specification itself can be incorrect or might not cover all cases. For instance, consider the A320 runway overrun in Warsaw in September 1993. Because of a certain combination of parameters (not covered in the specification)

the breaks and thrust reversers of the aircraft could not be activated immediately after touchdown (Ladkin 2000). The system nonetheless worked according to its specification.

Thus, in our field, the results of any new tool should be treated with extreme caution until at least one additional, independent implementation is available that yields analogous results. Furthermore, such an independent alternative implementation may also reveal errors in the specification/theory the tool is based upon. An example for this is the detection of an incorrect Hastings ratio calculation for Bayesian inference (Holder et al. 2005) which was unraveled in the course of such an independent implementation effort. We believe that this strategy of comparing the results of independent implementations (e.g., PHYML, IQ-Tree, RAxML for Maximum Likelihood or ExaBayes, MrBayes, PhyloBayes for Bayesian inference) represents a valuable approach to increasing our confidence regarding the correctness of these tools. In two independently developed tools for detecting terraces in phylogenetic tree space we directly applied this approach (Biczok et al. 2017).

In contrast to this, community projects such as R have been very successful, but R also represents a single point of failure. That is, errors in R core modules may have a more dramatic downstream impact than in MrBayes or RAxML, for instance. To this end, we advocate redundancy as *the* mechanism for increasing confidence about correctness.

Policy Issues

The 16 codes we analyzed have accumulated >90,000 citations (not including all papers describing updated versions) based on Google Scholar to date. One may argue that the amount of funding used to generate papers using these codes is disproportional to the amount of funding spent for maintaining and improving these codes, given the catastrophic effects that potential programming or conceptual bugs can have on the published results.

There is a clear lack of sustainable funding for programmers that could maintain and improve the codes developed by principal investigators or students that leave academia after their PhD. Firstly, one is limited by university or public sector salary schemes which are too low to hire outstanding programmers. Secondly, current funding schemes do not allow for hiring programmers on unlimited time contracts.

One may consider to allocate temporal funding for redesigning scientific codes to increase maintainability if they rapidly accumulate citations. This could be extended to funding several independent redundant implementations of emerging models and methods. The cost for this is small compared with the potential gains in quality and probability of code correctness.

Another problem is that there is insufficient funding for scientific software development per se. Numerous funding bodies do not consider scientific software development as being "real" research and it is thus extremely hard to obtain financial support. Ironically, a larger number of funded research projects (e.g., a search for the co-occurrence of the terms "phylogenetic" and "Deutsche Forschungsgemeinschaft" yields ~17,800 results in Google Scholar) relies on the availability of such tools.

Thus, due to the steadily increasing reliance on computational tools, we believe that novel funding schemes are required to develop new tools as well as improve quality and correctness of existing software. Moreover, the user community must be aware of the fact that, while current tools are freely available, they are developed on a best-effort basis only. There is a plethora of error sources, given that we simply do not have the time nor the resources to implement them properly and occasionally completely redesign them.

Alternatively, one may consider a commercial approach and raise license fees that could be used for providing support and maintenance. One disadvantage of this is that researchers from developing countries may not be able to afford the licenses. In addition, based on our experience with selling nonacademic licenses for the PEAR software (Zhang et al. 2014), license management can be time-consuming. Other potential licensing models include crowd-funding, pay-what-you-want strategies, or offering basic, free and advanced, non-free versions of a tool (e.g., including a graphical user interface).

Conclusion

We have presented an initial and simple software engineering quality assessment of widely used evolutionary biology software. We show that by using simple techniques and tools the quality of existing software could already be improved. We also provide a list of best practices for future software development projects. Furthermore, we address issues and provide real-world examples pertaining to numerical reproducibility (or lack thereof) to increase awareness about these issues in the user community. One must also keep in mind that, given the NGS data tsunami, there is a clear trade-off between program performance and maintainability. Programs like RAxML, that explicitly use vector intrinsics for maximum performance on standard laptop/server processor architectures, are substantially harder to maintain. As a consequence of this increased complexity, they are more error-prone than a straightforward naïve implementation of Felsenstein's pruning algorithm (Felsenstein 1981).

Further, we emphasize that the current and rather worrisome state of widely used software in our field is not the fault of the developers, but due to a substantial lack of sustainable funding for software development, improvement, maintenance, and support. This is especially true if one considers the disproportion between funding spent for generating the data with respect to funding spent for improving the quality of software for analyzing these data. We also make suggestions on how journals, editors, and reviewers could take measures for improving software engineering quality in the course of the review process. Furthermore, the independent development of software by different teams and the comparison of the results can substantially contribute to identifying correctness and not merely quality issues as we discuss them here.

We are convinced that, in the times of long and complex NGS data analysis pipelines with an ever increasing number of

components, software engineering quality issues are becoming critical to the success of the field. Thus, as long as there are no additional efforts on improving software engineering quality, and given the current unsatisfactory quality of tools, users should not treat evolutionary analysis tools as black boxes, but rather as potential Pandora's boxes. Apart from improving software engineering quality, we also need to invest more effort into the systematic validation of the results produced by our codes in the future.

Supplementary Material

Supplementary data are available at *Molecular Biology and Evolution* online.

Acknowledgments

We wish to thank Volker Springel, Bastien Bousseau, and Tracy Heath for suggestions and discussions regarding this project. We would also like to thank our software engineering colleague Ralf Reussner at KIT for insightful discussions. We are particularly grateful to Mark Holder for extremely useful suggestions and comments on an earlier version of this manuscript. We wish to thank Stephane Guindon and Fredrik Ronquist for their reviews of the initial version of this manuscript. This work was funded by the Klaus Tschira Foundation.

References

Abdelmalek NN. 1971. Round off error analysis for Gram–Schmidt method and solution of linear least squares problems. *BIT Numer. Math*. 11(4):345–367.

Barone L, Williams J, Micklos D. 2017. Unmet needs for analyzing biological big data: a survey of 704 nsf principal investigators. *PLoS Comput Biol* 13(10):e1005755.

Biczok R, Bozsoky P, Eisenmann P, Ernst J, Ribizel T, Scholz F, Trefzer A, Weber F, Hamann M, Stamatakis A. 2017. Two C++ libraries for counting trees on a phylogenetic terrace. *bioRxiv*. https://www.biorxiv.org/content/early/2017/11/02/211276.

Briand LC, Wüst J, Ikonomovski SV, Lounis H. 1999. Investigating quality factors in object-oriented designs: an industrial case study. In: Proceedings of the 21st International Conference on Software Engineering, ACM. p. 345–354.

Briand LC, Wüst J, Daly JW, Porter DV. 2000. Exploring the relationships between design measures and software quality in object-oriented systems. *J. Syst. Softw*. 51(3):245–273.

Casalnuovo C, Devanbu P, Oliveira A, Filkov V, Ray B. 2015. Assert use in github projects. In: Proceedings of the 37th International Conference on Software Engineering – Volume 1, ICSE '15, Piscataway (NJ): IEEE Press. p. 755–766.

Chen TY, Cheung SC, Yiu SM. 1998. Metamorphic testing: a new approach for generating next test cases. Technical report, Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong.

Chen TY, Ho JW, Liu H, Xie X. 2009. An innovative approach for testing bioinformatics programs using metamorphic testing. *BMC Bioinformatics*. 10(1):24.

Drummond AJ, Rambaut A. 2007. BEAST: Bayesian evolutionary analysis by sampling trees. *BMC Evol. Biol*. 7(1):214.

Felsenstein J. 1981. Evolutionary trees from DNA sequences: a maximum likelihood approach. *J. Mol. Evol*. 17(6):368–376.

Fletcher W, Yang Z. 2009. INDELible: a flexible simulator of biological sequence evolution. *Mol. Biol. Evol*. 26(8):1879–1888.

Giannoulatou E, Park S-H, Humphreys DT, Ho JW. 2014. Verification and validation of bioinformatics software without a gold standard: a case study of BWA and bowtie. *BMC Bioinformatics*. 15(Suppl 16):S15.

Goldberg D. 1991. What every computer scientist should know about floating point arithmetic. *ACM Comput. Surv*. 23(1):5–48.

Grimm GW, Renner SS, Stamatakis A, Hemleben V. 2006. A nuclear ribosomal DNA phylogeny of acer inferred with maximum likelihood, splits graphs, and motif analysis of 606 sequences. *Evol. Bioinform. Online* 2:7.

Guindon S, Dufayard J-F, Lefort V, Anisimova M, Hordijk W, Gascuel O. 2010. New algorithms and methods to estimate maximum-likelihood phylogenies: assessing the performance of PhyML 3.0. *Syst. Biol*. 59(3):307–321.

Heath TA, Huelsenbeck JP, Stadler T. 2014. The fossilized birth–death process for coherent calibration of divergence-time estimates. *Proc. Natl. Acad. Sci. U. S. A*. 111(29):E2957–E2966.

Hoare CAR. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12(10):576–580.

Holder MT, Lewis PO, Swofford DL, Larget B. 2005. Hastings ratio of the LOCAL proposal used in Bayesian phylogenetics. *Syst. Biol*. 54(6):961–965.

Hudson RR. 2002. Generating samples under a Wright–Fisher neutral model of genetic variation. *Bioinformatics* 18(2):337–338.

Jarvis ED, Mirarab S, Aberer AJ, Li B, Houde P, Li C, Ho SY, Faircloth BC, Nabholz B, Howard JT. 2014. Whole-genome analyses resolve early branches in the tree of life of modern birds. *Science* 346(6215):1320–1331.

Johnson SC. 1977. *Lint, a C program checker*. Citeseer.

Juergens E, Deissenboeck F, Hummel B, Wagner S. 2009. Do code clones matter? In: IEEE 31st International Conference on Software Engineering, 2009. ICSE 2009. IEEE. p. 485–495.

Kamali AH, Giannoulatou E, Chen TY, Charleston MA, McEwan AL, Ho JW. 2015. How to test bioinformatics software? *Biophys. Rev*. 7(3):343–352.

Katoh K, Standley DM. 2013. MAFFT multiple sequence alignment software version 7: improvements in performance and usability. *Mol. Biol. Evol*. 30(4):772–780.

Kozlov AM, Aberer AJ, Stamatakis A. 2015. Examl version 3: a tool for phylogenomic analyses on supercomputers. *Bioinformatics* 31(15):2577–2579.

Kumar S, Dudley J. 2007. Bioinformatics software for biologists in the genomics era. *Bioinformatics* 23(14):1713–1717.

Ladkin PB. 2000. Causal reasoning about aircraft accidents. In: Computer Safety, Reliability and Security, Berlin, Heidelberg: Springer. p. 344–360.

Lawall J, Laurie B, Rydhof Hansen R, Palix N, Muller G. 2010. Finding error handling bugs in openssl using coccinelle. In: Proceeding of the 8th European Dependable Computing Conference, EDCC 2010, Valencia, Spain. p. 191–196.

Leprevost FdV, Barbosa VC, Francisco EL, Perez-Riverol Y, Carvalho PC. 2014. On best practices in the development of bioinformatics software. *Front. Genet*. 5:199.

Li R, Yu C, Li Y, Lam T-W, Yiu S-M, Kristiansen K, Wang J. 2009. SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics* 25(15):1966–1967.

Li Z, Tan L, Wang X, Lu S, Zhou Y, Zhai C. 2006. Have things changed now? An empirical study of bug characteristics in modern open source software. In: Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability, ASID '06, New York (NY): ACM. p. 25–33.

Löytynoja A, Goldman N. 2005. An algorithm for progressive multiple alignment of sequences with insertions. *Proc. Natl. Acad. Sci. U. S. A*. 102(30):10557–10562.

Lu S, Li Z, Qin F, Tan L, Zhou P, Zhou Y. 2005. Bugbench: Benchmarks for evaluating bug detection tools. In: Workshop on the Evaluation of Software Defect Detection Tools.

McCabe TJ. 1976. A complexity measure. *IEEE Trans. Softw. Eng*. SE-2(4):308–320.

Misof B, Liu S, Meusemann K, Peters RS, Donath A, Mayer C, Frandsen PB, Ware J, Flouri T, Beutel RG, et al. 2014. Phylogenomics resolves the timing and pattern of insect evolution. *Science* 346(6210):763–767.

Myers GJ, Sandler C, Badgett T. 2011. The Art of Software Testing. Hoboken, New Jersey: John Wiley & Sons.

Nielsen R, Williamson S, Kim Y, Hubisz MJ, Clark AG, Bustamante C. 2005. Genomic scans for selective sweeps using SNP data. *Genome Res*. 15(11):1566–1575.

Notredame C, Higgins DG, Heringa J. 2000. T-Coffee: a novel method for fast and accurate multiple sequence alignment. *J. Mol. Biol*. 302(1):205–217.

Rambaut A, Grass NC. 1997. Seq-Gen: an application for the Monte Carlo simulation of DNA sequence evolution along phylogenetic trees. *Comput. Appl. Biosci.: CABIOS*. 13(3):235–238.

Robinson D, Foulds L. 1981. Comparison of phylogenetic trees. *Math. Biosci*. 53(1–2):131–147.

Ronquist F, Teslenko M, van der Mark P, Ayres DL, Darling A, Höhna S, Larget B, Liu L, Suchard MA, Huelsenbeck JP. 2012. MrBayes 3.2: efficient Bayesian phylogenetic inference and model choice across a large model space. *Syst. Biol*. 61(3):539–542.

Rother K, Potrzebowski W, Puton T, Rother M, Wywial E, Bujnicki JM. 2012. A toolbox for developing bioinformatics software. *Brief. Bioinf*. 13(2):244–257.

Simpson JT, Wong K, Jackman SD, Schein JE, Jones SJ, Birol I. 2009. ABySS: a parallel assembler for short read sequence data. *Genome Res*. 19(6):1117–1123.

Springel V. 2005. The cosmological simulation code gadget-2. *Month. Not. R. Astron. Soc*. 364(4):1105–1134.

Stamatakis A. 2014. RAxML version 8: a tool for phylogenetic analysis and post-analysis of large phylogenies. *Bioinformatics* 30(9):1312–1313.

Wilson G, Aruliah D, Brown CT, Hong NPC, Davis M, Guy RT, Haddock SH, Huff KD, Mitchell IM, Plumbley MD. 2014. Best practices for scientific computing. *PLoS Biol*. 12(1): e1001745.

Wilson G, Bryan J, Cranston K, Kitzes J, Nederbragt L, Teal TK. 2017. Good enough practices in scientific computing. *PLoS Comput. Biol*. 13(6): e1005510.

Yamamoto Y, Nakatsukasa Y, Yanagisawa Y, Fukaya T. 2015. Roundoff error analysis of the choleskyqr2 algorithm. *Electron. Trans. Numer. Anal*. 44:306–326.

Yang Z. 2007. PAML 4: phylogenetic analysis by maximum likelihood. *Mol. Biol. Evol*. 24(8):1586–1591.

Yang Z, Rannala B. 2010. Bayesian species delimitation using multilocus sequence data. *Proc. Natl. Acad. Sci. U. S. A*. 107(20):9264–9269.

Zhang J, Kobert K, Flouri T, Stamatakis A. 2014. Pear: a fast and accurate illumina paired-end read merger. *Bioinformatics*. 30(5):614–620.