Walking the talk: adopting and adapting sustainable scientific software development processes in a small biology lab.

Michael R. Crusoe[1]
C. Titus Brown[2,1*]
1 Microbiology and Molecular Genetics,
2 Computer Science and Engineering,
Michigan State University, East Lansing, MI, USA
* Corresponding author: ctb@msu.edu

September 2013

Abstract

The khmer software project implements both research and production functionality for large-scale sequence analysis. The project's development is driven by a small lab with only one full-time developer (MRC), as well as several graduate students and a professor (CTB) who contribute regularly to research features. Here we describe our efforts to bring better design, testing, and more open development to the khmer software project, which has just released v1.0.

1 Introduction

Computational tools for analyzing large volumes of sequencing data have become increasingly necessary over the last decade. The growth of sequencing capacity and the associated expansion of scientific problems being studied with sequencing is driving the development of many new tools, both for handling data on large scales and to address new and different biological problems.

The khmer software was born from a need to more scalably analyze short (20-30 character) words, or "k-mers", in large DNA sequencing data sets. The use of k-mers in DNA sequence analysis is common because they can be easily hashed, counted and compared within and between data sets. However, as data sets have grown in size, approaches to analyzing k-mers have fallen behind the memory and compute scaling curves. khmer provides several research and production functions, including approximate k-mer counting using a CountMin Sketch [**?**], an implementation of a compressible k-mer connectivity graph [**?**], and a streaming lossy compression algorithm for large data sets [**?**].

We have developed the khmer software as an open source project since the beginning: the software is under the BSD license, and we use GitHub for development, code review, release tagging, and other development activities. We have made a wide variety of tutorials and user documentation available, as part of the khmer project itself and also as part of a range of workshops. Adoption of khmer not only by its utility in addressing otherwise difficult or intractable problems, but also by CTB's blogging, research preprints and publications, and presentations. We now estimate that we have over 500 users of the khmer software itself, and the algorithms and approaches initially implemented in khmer have been adapted and incorporated to other software packages.

Our main challenge in developing khmer has been to simultaneously support an energetic research program in bioinformatics while working towards a stable and reliable software product. Below, we discuss our experience in navigating a course towards a sustainable small-lab software project.

2 Background

khmer grew out of specific analysis needs, and was developed primarily on startup funding and as part of a USDA grant. Its development has led to at least two additional grants, including the NIH BIG DATA grant that now supports MRC[1]. Over its lifetime khmer has had 15 different contributors, with five currently active. The code consists of approximately 12.2k lines of C and C++ code, with scripts and tests written in Python (6.6k lines of code).

The software was initially written in CTB for other purposes during his graduate work at Caltech, and then extended so far as to be almost entirely rewritten for research in his faculty position at Michigan State University. By July of 2013, when MRC started, the software had its current level of functionality, but we faced a number of specific challenges.

- First, we had no formal development model: no code review, no formatting requirements, no continuous integration, and no API stability requirements. This meant that we were constantly in a state of uncertainty about khmer's quality.
- Second, our developers were at a variety of experience levels: some were expert computational biologists with little to no programming experience, while others were experienced open source software developers with little to no computational biology background. This meant that we could not confidently rely on good domain understanding or good software development hygiene from any one developer.
- Third, like many bioinformatics projects, khmer was both *research* and *production* software: our lab was constantly extending khmer in new directions, while we and others were applying its existing functionality to pressing biology research problems. While this is a traditional problem of software development on long-running projects, the problem was exacerbated because long-term planning was impractical given the high rate of technical innovation in sequencing data generation.
- Fourth, khmer exists within an ecosystem of tools. khmer itself primarily acts as a filter for sequence data, which is generated in particular formats by upstream tools and is then consumed in those same formats by downstream tools. We had no systematic testing of khmer within its larger ecosystem.

Collectively, these challenges meant that khmer software development was probably not sustainable: without significant investment in software engineering, either research development would falter in the face of increasingly high maintenance demands, or the stable functionality would start to deteriorate, or both.

To address these challenges, CTB secured three years of NIH funding through the 2012 BIG DATA competition, and hired a software developer, MRC.

3 Initial Evaluation

To guide our development of a rational software development process, we applied the Software Sustainability Institute's Criteria-based assessment checklist [5] to the khmer project in September 2013 and shared the results with the community [3]. The summary from that report was grim: khmer met 17 of 42 (or 40%) of of the SSI's criteria for Usability, and 43 of 119 (or 36%) of the criteria for Sustainability, for an overall fulfillment of 43 of the 119 items, or 36%.

The low fulfillment of the SSI's criteria is slightly embarrassing for the group for several reasons: CTB is a collaborator and co-author with SSI; quotes from him are featured in case study presentations [2]; he is cited as someone who is doing things the correct way [4]; and he regularly criticizes the field for poor software process. This is the point, however: because these goals were highly valued within the lab, and because the software filled a novel niche, CTB successfully argued for resources to address these issues [1]. Making significant progress towards these ends isn't merely a matter of budgets and hiring: even with a full time hybrid software engineer/bioinformatician on the team it is not straightforward to prioritize infrastructure work, let alone navigate between the core research work, user support, community engagement, and collaborations.

[1] @@@link

4 Releasing version 1.0

On April 1st, 2014, we released khmer 1.0. While by no means a finished product, we believe we are on a much more sustainable development path. Version 1.0 contains XXX, YYY, ZZZ. Below, we discuss in more detail.

4.1 Development standards and semantic versioning

We instituted a number of development standards, including coding styles and requirements for backwards compatibility. Our goal was to have explicit written requirements that would inform new contributors, with or without significant prior programming experience.

Uniformity of coding styles helps maintain code readability and enables easier code review. The specific choice for coding style was made somewhat arbitrarily, largely to avoid bikeshedding discussions: the important goal was to have *some* coding standard. And, since automatic reformatting software is freely available, editor configuration is not a significant burden on developers. For C++, we chose the "One True Brace Style" of astyle for indentation and bracing. For Python, we settled on the default PEP8 standard, for which several checking and reformatting tools exist.

We also imposed a backwards compatibility requirement on our command line scripts. While we did not want to stabilize the Python or C++ API because we are actively changing khmer internals, we felt that our command line scripts were sufficiently stable to require no backwards-incompatible changes on 1.0.

We have therefore committed to *semantic versioning* for the command line scripts that come with khmer. This imposes a three-tiered versioning system: for patch version number changes (khmer v1.0.x), only minor bug fixes and documentation updates are allowed; for minor version number changes (khmer v1.x), backwards compatibility of the command line scripts must be maintained; and, should we choose to break backwards compatibility, we would need to make a major version number change (khmer v2).

The importance of semantic versioning is that it allows developers, documentation writers, sysadmins, and package managers to rely on specific behavior from a range of versions. Of particular importance, pipeline developers and users can rely on stable behavior from minor releases. We expect this to make khmer a more reliable member of the sequencing analysis software ecosystem.

4.2 Continuous integration

Continuous integration ensures that test code is executed regularly on standard platforms. While developers are expected to commit code with no failing tests, often developers do not have convenient access to many different operating system and install environments. Continuous integration frees individual developers from having to execute their tests manually by automating the process Our continuous integration system, built on top of Jenkins and running on a Rackspace donated Linux server and an internal Mac OS X machine, also runs style checkers and reports code coverage summaries.

The most important application of continuous integration for us has been automated checking of merge requests prior to code review or merge. This automatically ensures a base quality of code.

@cite chapter on CI.

4.3 Integrated code coverage analysis

Code coverage analysis is an important part of software development: statement coverage, or how many lines of code are executed in some way by unit and functional tests, is relatively simple to calculate and identifies untested regions of code.[2] While khmer had several hundred tests by the time MRC started working with it, the tests were all at the Python level and we had no estimate of how well they covered the C++ code base.

Combined C++ and Python code coverage was instituted in October 2013 and we were pleasantly surprised to find that over 80% of the khmer codebase was executed in the tests. Since October we have increased the code coverage percentage to over 90%. This number is now calculated on every pull request

[2]Note that while executed code is not guaranteed to be correct, code that is not executed by tests is certainly not tested, so high code coverage is a necessary but not sufficient condition for thorough testing.

(see below) and significant decreases are flagged as "unhealthy" in our continuous integration system.

4.4 Code contribution process and code review

While code review is an important part of ensuring that only "good" code and feature are included in a project, it is typically very time consuming to do systematic code reviews. In order to scale our development process to more contributors while enabling code review, we adopted the "GitHub Flow" model of code review (@@citegithub flow). In this model, changes are developed on an independent branch of code; this branch of code is linked to the main development repository via a "pull request", which is an ongoing summary of changes together with free text discussion. When the developer feels that the changes are ready to be merged, they request a formal review, for which we have instituted a checklist; this checklist includes test coverage and coding style analysis, documentation review, and compatibility checking.

Our expectation is that this more formal but still lightweight development process will encourage contributions and also serve as a training and education process for less experienced developers. By making our developer contribution requirements explicit, we may also serve as a guide for other bioinformatics software projects.

4.5 Citation information

Scientific funding and reputation depends significantly on citations. For both algorithms and software, citations demonstrate usage, utility, and impact. We therefore added explicit citation guidelines in two places: first, in the CITATION file at the top of the distribution, and second, at the top of every script. We ask that users cite not only the software itself (via a software paper), but also the papers describing the algorithms implemented.

@include some discussion of double citation concern.

4.6 Integration and acceptance testing

An ongoing concern for khmer is how well our software integrates with other packages. Because khmer primarily interacts with the output of upstream software, and the output of khmer is then fed into downstream software, we need to take into account a larger software ecosystem. Moreover, there are few real data format standards exist in this area: the sequencing companies that generate the source data are notoriously arbitrary in changing their output formats, and developers of other packages may introduce format changes intentionally (through feature extension) or unintentionally (through bugs). Standardization itself is probably a futile approach: while we may expect A, C, G, and T to remain the primary characters in DNA sequence representations, the formats for uncertainty and annotation are evolving with sequencing technology, which in turn is changing quickly.

We therefore have instituted *acceptance testing* to ensure that khmer works with at least some upstream and downstream software packages. Our acceptance tests for khmer 1.0 take a subset of data through quality control, error trimming, digital normalization, and assembly; at the end we ensure that we obtain approximately the expected results. We have been greatly aided in developing acceptance tests by our standard "protocols" for sequence analysis: our acceptance tests go through the first three parts of the Eel Pond mRNAseq protocol (http://khmer-protocols.readthedocs.org/).

Acceptance testing proved to be extremely important in the release process. Four different bugs having to do with installation and command-line parameter handling were discovered in the 48 hours before release; these bugs generally had to do with common command line cases that were not readily testable at the unit and functional level.

5 Persistent Challenges in Research Software Development

@CTB review and update.

In the long term, we expect to face three major challenges in continuing to develop khmer.

First, we need to show the value of the process, so that we and others can convince colleagues and funding agencies that software development processes should be a first-class participant in publication and

funding considerations. This will almost certainly have to be done by doing a better (faster, higher quality) job of tackling core scientific and biological questions, and will need to be measured in publications and citations.

Second, we must continually reevaluate how much and where to refactor and test. In the face of changing input data (due to instrument and experimental protocol changes), different expectations for output (bioinformaticians invent a new format every 5 minutes on average), competing algorithms with poor replicability, etc., we believe we could spend 100% of our time on quality control without developing anything truly new. However there is little or no reward in academia for merely continuing to produce functioning software: software maintenance grants are few and far between. So our core software maintenance efforts must instead be directed at enabling us to agilely tackle research problems, i.e. focus on issues that we don't yet understand or for which we don't have much intuition. While this may seem like a standard software engineering problem, we believe that the nature of pure research may lead to additional challenges here.

Third, we face systemic and severe cultural challenges in terms of recruiting software developers and researchers. Inevitably new lab members are undertrained in most of what we do, including testing, version control, good computational hygiene, data science, and/or the domain of biology. These are a lot of subjects to train new people in, and we have yet to establish an effective lab culture. On the converse side, of course, we expect lab graduates to be increasingly employable in both academia and biotech; moreover, the lab reputation of caring about such things has started to attract people who are already somewhat better trained.

6 A brief dialogue

@CTB refactor into discussion.

MRC: What particular challenges do you think life scientists face in making the software artifacts of the research process reusable and sustainable?

CTB: The two primary challenges are cultural (which was expected) and technical (which was unexpected). By cultural, I mean that there is virtually no culture of computational reproducibility or software development in biology, which makes it hard to discuss much less justify. Technically, the infrastructure and tools for enabling reproducibility are still quite young and do not fit the needs of subdomains terribly well. For example, I thought that other fields would have tools for provenance and workflow tracking that we could easily adapt; this is simply not the case. In fact, our most effective set of tools has emerged from open source and data science work, including the excellent Python and GitHub communities, IPython Notebook, and cloud computing infrastructure.

@talk about importance of software; novelty squared.

future plans?

Acknowledgments

MRC has been funded by AFRI Competitive Grant no. 2010-65205-20361 from the USDA NIFA and is now funded by the National Human Genome Research Institute of the National Institutes of Health under Award Number R01HG007513; both under C. Titus Brown.

References

[1] C. Titus Brown. The future of khmer (2013 version). `http://ivory.idyll.org/blog/the-future-of-khmer-2013-version.html`, 2013.

[2] Steve Crouch. Institute case studies: From consultancy projects to case studies. `http://www.software.ac.uk/attach/CaseStudies.pdf`, 2012.

[3] Michael R. Crusoe. Criteria-based assessment of the khmer suite. `http://goo.gl/MZGGhc`, 2013.

[4] Ian Gent. The recomputation manifesto. `http://www.software.ac.uk/blog/2013-07-09-recomputation-manifesto`, 2013.

[5] Mike Jackson, Steve Crouch, and Rob Baxter. Software evaluation guide. http://www.software.ac.uk/resources/guides-everything/software-evaluation-guide, 2013.