How to professionally develop reusable scientific software — and when not to

A critical challenge in scientific computing is balancing developing high quality software with the need for immediate scientific progress. We present a flexible approach that emphasizes writing specialized code that is refactored only when future immediate scientific goals demand it. Our lazy refactoring technique, which calls for code with clearly defined interfaces and sharply delimited scopes to maximize reuse and integrability, helps reduce total development time and accelerates the production of scientific results. We offer guidelines for how to implement such code, as well as criteria to aid in the evaluation of existing tools. To demonstrate their application, we showcase the development progression of tools for particle simulations originating from the Glotzer Group at the University of Michigan. We emphasize the evolution of these tools into a loosely integrated software stack of highly reusable software that can be maintained to ensure the long-term stability of established research workflows.

Carl S. Adorf
Department of Chemical Engineering, University of Michigan, Ann Arbor

Vyas Ramasubramani
Department of Chemical Engineering, University of Michigan, Ann Arbor

Joshua A. Anderson
Department of Chemical Engineering, University of Michigan, Ann Arbor

Sharon C. Glotzer
Department of Chemical Engineering, University of Michigan, Ann Arbor

1) Both authors contributed equally to this article.

As computational science continues to play a rapidly growing role across nearly all scientific disciplines, quality scientific software becomes ever more critical to research productivity. The development of such software is impeded by the fact that most scientific software is developed by non-experts for whom the primary product is not the software, but the resulting science. The existing incentive structure, which is geared towards rapid publication of scientific results, tends to motivate the creation of single-use software. In addition to being highly targeted at a specific application, single-use software often also disregards established software development best practices such as proper version control and documentation, impeding its reuse for future studies and ultimately hindering long-term scientific progress.

Software solutions for many-body simulations present a useful case study of this development pattern. Over time, the range of scales and physical phenomena of interest have led to the creation of numerous packages...
with overlapping yet distinct feature sets. Despite this fragmentation, however, certain packages have attract-
ed much larger user communities than others, suggesting a slow progression towards specific reusable solu-
tions.

We present an approach, lazy refactoring, for accelerating this progression while ensuring that code devel-
opment is always targeted at an immediate scientific objective. This approach—especially suited for re-
searchers, who need to reconcile sustainable software development with the need to make immediate
scientific progress—advocates that initial development should always lead to single-use code, but this code is refactored into a reusable solution as soon as two further uses for it are found. We discuss how software
standards such as modular design and well-defined interfaces, tools such as VCS and CI, and code proper
documentation help facilitate lazy refactoring. With this context, we offer a concrete methodology to help
determine when and how to write new code.

As an example of lazy refactoring, we present the development progression of software developed by the
Glotzer Group at the University of Michigan. We are a diverse, collaborative research group comprised of
30+ graduate students, post docs, and research scientists from chemical engineering, materials science, phys-
ics, and other disciplines. Roughly half our members join the group with little to no simulation experience,
and very few have developed professional quality code prior to joining. As such, we are a representative
sample of the computational research community within academia. We show how lazy refactoring evolved
alongside our development efforts and demonstrate its use to create a domain-specific, reusable, and loosely
integrated software stack. We close with a brief discussion of how we train group members to foster the pre-
sented techniques.

DEVELOPING COMPUTATIONAL SOLUTIONS

The goal of lazy refactoring is to minimize the total resources used on tool development for a computational
problem within the context of a preexisting software ecosystem, which refers to the set of all available soft-
ware. While much of this code will come in the form of clearly defined, well-documented, and consistently
maintained software packages, the ecosystem will also contain harder to reuse special-purpose codes that
may nevertheless be useful beyond their original intent. This second kind of software, commonly referred to
as a prototype, generally lacks clean interfaces and documentation, and does not have explicit names or re-
lease versions. While packages are easily reusable, prototypes offset this benefit by being faster to implement
and requiring less effort to maintain. Borrowing from well-established agile software development tech-
niques such as extreme programming (XP), lazy refactoring ensures that the ability to respond rapidly to
changing requirements and maintaining working software are prioritized over planned design and compre-
hensive documentation. One critical difference here is that the programmer of scientific software aimed at
solving a particular scientific problem is usually also the only customer.

In principle, total development efforts are minimized by using existing code whenever possible, and, when
new code is required, expending effort to make code reusable only when reuse is expected. From this per-
spective, solving a computational problem starts by decomposing it into components that each require a
software solution. This process helps identify which parts are already solved within the ecosystem, which
ones can be addressed by adapting existing tools, and which ones require entirely new code. Then, the miss-
ing components can be assessed to determine which ones merit developing reusable packages and which ones
should be developed as prototypes. We use the term adapter to refer to code whose primary purpose is to
interface between other packages.
Figure 1: This figure illustrates the code development progression associated with solving two related computational problems. Problem A is solved using packages a and b in combination with some adapter code and the prototype code base x. Problem B is determined to overlap with problem A, and large portions of it are already solved by packages b and c. There are two alternatives for addressing the remaining software needs for problem B. The first alternative is to reuse x and develop prototype y to fill the gap (Alternative A). The second approach is to refactor prototype x into package d, which interfaces with packages a, b, and c and fully solves problem B (Alternative B).

In practice, however, making an optimal decision about when to develop reusable code is generally impossible. The problem scope may be ill-defined and future project plans change, invalidating total resource estimates and making it difficult to determine which components are worth developing as packages. In this light, it is clear that developing reusable code packages any time reuse is foreseen is often an inefficient use of resources.

Instead, we recommend the more agile lazy refactoring approach, in which all missing code is developed as a prototype in the most convenient manner possible, e.g., in Python. Refactoring of this code should always be motivated by its imminent application to new problems, but refactoring should be preferred over new prototype development once such applications are identified. Specifically, we advocate following the Rule of Three: a prototype should be refactored as soon as a third application is found. We illustrate this approach in Figure 1, where there are two alternatives to address the additional software needs of Problem B after previously solving Problem A. If no prototypes exist, the problem should be solved with rapid development of prototypes x and y, tailored to the specific problem (Alternative A). If two or more prototypes already exist, then the gap should be closed by refactoring these into package d (Alternative B). We argue that, on average,
this approach minimizes total software development effort since the resources required for the design, implementation and maintenance of software packages are only expended once the future reuse potential is sufficiently apparent.

The advantages of lazy refactoring are manifold:

1. Total resource investment into solving one problem is minimized.
2. Refactoring a prototype into a package is usually easier than writing a package from scratch because it postpones developing interfaces and defining the scope until the problem is better understood.
3. The package can be validated against the original prototype.
4. The decision to refactor can account for actual projects that arise rather than attempting to estimate future reusability.

Lazy refactoring does carry significant risks: key personnel involved with the original development may no longer be available for refactoring, or the prototype may grow so large and opaque that refactoring itself presents a significant barrier. Therefore, the Rule of Three should be applied rigorously such that refactoring occurs in a timely manner and the original developers can be heavily involved in the refactoring process. This method requires that some quality controls be imposed on even initial prototypes to ensure that refactoring remains possible.

PRINCIPLES, TOOLS AND PRACTICES

In this section, we expound upon several principles, development tools and best practices that aid in effectively applying lazy refactoring. As we will show, these principles, tools, and practices are also useful barometers for evaluating preexisting software solutions, as will be shown later.

Principles

Modularity, the core design principle that we advocate, is a well-known standard enshrined in the UNIX philosophy that naturally leads to the development of tools with clearly delimited scopes and well-defined Application Programming Interfaces (APIs), each of which confer significant benefits. Code with limited scope can be developed more quickly, and it may preclude further time expenditures if the initial work proves sufficient for the task at hand. The API-driven design model also ensures that different components can be easily used independently of one another, making it easier to generalize and refactor specific pieces or integrate them with other tools. This feature is especially crucial in science because specific research applications often require combining previously unrelated, highly domain-specific software.

The first step towards modularity is defining a clear division between software components. Defining these divisions immediately suggests the appropriate scopes for each component, the specific interoperability requirements, and the necessary API to meet these requirements. In section “The Glotzer Group Software Stack” we show how our group’s stack is largely composed of modular packages with well-defined APIs that employ standard data formats and operational paradigms for easy integration.

Our other guiding principle is the public release of software source code. Although some code may be privately maintained to preserve a competitive advantage in the short term, the need to validate scientific results and the need to integrate disparate software tools make the ability to view source code more valuable than ever. These are strong incentives for open-source scientific software development.

Tools and Best Practices

We employ a number of tools and best practices to ensure that our code adheres to the principles expressed in the previous section. Version Control Systems (VCSs), which enable the robust management of a project’s evolution over time, are especially important in scientific code development due to its collaborative nature. VCSs enable scientists to work in parallel while maintaining stable, working versions of code at all times. This collaborative process is much better suited to the decentralized model of Distributed Version Control Systems (DVCSs), such as Git, than to the centralized approaches of traditional VCSs like Subversion (SVN). The benefits of a controlled, shareable central repository can be recovered by hosting code on public repositories such as GitHub, Bitbucket, or GitLab.
Some code quality checks can be automated using code linters, such as Flake8 for Python, splint for C, or Cppcheck for C++. Higher level checks can solicit input by using, for instance, pull request-based workflows to require human reviews before new code is added to an existing code base. Code should include unit and integration tests, which increase confidence in code correctness and reduce the likelihood of introducing breaking changes. Code should also employ Continuous Integration (CI), the automatic execution of tests to ensure that any errors or issues are caught before spreading to stable code versions. Many CI services, such as Jenkins, CircleCI, and Bitbucket Pipelines, integrate with repository hosting services and are free to publicly available repositories, further incentivizing open-source development.

To maximize reusability, software must be widely disseminated and painless to install. At the very least, source code should be publicly available, but standardizing modes of distribution through, e.g., package managers, is the preferred mode of making the software readily available. For example, most open-source packages published by our group are distributed via the PyPI and Anaconda cloud repositories. For distribution on specialized High Performance Computing (HPC) environments, we currently host Docker images on the Docker container hub, which we deploy using Singularity on environments such as the TACC Stampede2, PSC Bridges, and SDSC Comet clusters available through XSEDE.

Software must come with effective documentation, which includes both guidance for high-level usage and more detailed information on the package’s various components such as its API. Documentation quality can be greatly improved by taking into account user feedback, which can be obtained through surveys and focus group sessions along with more asynchronous mechanisms like issue trackers, mailing lists, and chat rooms. As with CI, online documentation hosting services such as ReadTheDocs are free for open-source software to simplify the publishing of high-quality documentation.

Finally, software must be appropriately licensed. Terms and conditions for using unlicensed code may be unclear, so software (whether open- or closed-source) must have licenses that permit reuse. For open-source software, for instance, the Open Source Initiative (OSI) approves certain licenses as providing sufficiently free usage and modification. Our group typically licenses software under the BSD 3-clause or the MIT license.

APPLYING LAZY REFACTORING

While we have described an agile approach to code development for a research problem, to this point our formulation has been purposely abstract, providing only a conceptual description and high-level guidelines. We now describe the practical application of lazy refactoring to solving a particular problem. Using the ideas espoused by the previous section as a guide, we identify specific attributes by which existing software may be assessed and that should be incorporated into any new software. We then present a process for rigorously and systematically determining exactly when and how much new code development is merited, using the identified attributes to assess the usability and integrability of external code bases. Note that the needs of a typical project will evolve over its lifetime, and will therefore require applying this heuristic multiple times.

Critical Attributes

Any software that is considered for integration into a problem solution workflow should be assessed according to the following critical attributes:

1. **Scope**: Does it solve the problem at hand?
2. **Integrability**: How well does it integrate into the existing software stack, including external tools upon which this stack relies, and how much work would be required for integration?
3. **Stability**: Does it have a well-defined, static API, is it largely error-free, and is it likely to be maintained into the foreseeable future, or at the very least, over the project lifetime?
4. **Security**: Does it pose any security risks?

These attributes are discussed in detail in the following subsections.

Scope

In order for software to be useful, it must solve the problem at hand. Although this statement appears trivial, its simplicity hides some important nuances. A complete solution must satisfy, for instance, scalability and performance requirements. These factors are context-dependent; for example, simulations may need to scale
to leadership-class HPC platforms and run fast even for systems with millions of particles. Such considerations may disqualify a candidate code base that solves the problem in a limited set of cases.

Ease of integration

The next step is determining how easily the software can be integrated into existing workflows. As discussed earlier, license compatibility is a significant factor in this determination and may be a reason to prefer open-source software. Open-source software has the additional benefit that the code can be inspected and adapted if needed (see Section “Guidelines for package adaptation”).

In addition to license compatibility, software compatibility also encompasses modularity, data types, and file formats. In order for different components of a software pipeline to work together, they must communicate in some common language, i.e., they must be able to exchange data in a standardized manner. For example, all numerical analysis software implemented in Python should make use of NumPy, which is the de facto standard for arrays of numerical data within the Python ecosystem.

Stability

High quality software must also be sufficiently stable. This means that API and code paths are set, further dependencies are unlikely to be introduced, and existing features are provided with the implicit or explicit promise that they will continue to work. Furthermore, stable packages must be well-supported and have good documentation, and they should have active and responsive developers. Additional indicators of stability are active mailing lists, issue trackers, and forums.

Security

Last but not least is the consideration of security risks posed by potential external software solutions. For instance, any software that requires a privileged execution context may be problematic, especially in HPC environments. Additionally, software that communicates with internet servers, including cloud computing services, may be unsuitable for handling sensitive data. These services are commonly not compliant with rules and regulations implemented by the Health Insurance Portability and Accountability Act (HIPAA) in the U.S. and similar legislation in other countries.
Figure 2: This flow chart depicts the core elements of a basic decision tree for the development of workflows that solve a specific computational problem. In summary, we break down the problem into individual components that may or may not have existing solutions. Existing solutions are evaluated for use, and where appropriate, prototype solutions are either written or existing prototypes are refactored into packages. The decision on what solutions to include is largely based on the criteria outlined in "Applying Lazy Refactoring", which serve as basic guidelines for evaluating existing solutions for their integrability into the proposed workflow.

General Heuristic

We now formulate a decision tree to guide the software-related decisions involved in solving a scientific problem (see Figure 2). This process involves decomposing a problem into components that can be solved sequentially. A critical part of the process is the usage of preexisting partial solutions, which are often originally developed as single-use prototypes and are ripe for refactoring. Although such refactoring can be difficult, eventually it will pay dividends as the quality of the software ecosystem increases and software becomes easier to refactor and adapt. NumPy is one example of how two partial solutions, Numarray and Numeric, were consolidated into an improved package that shares both their strengths.

In some circumstances, writing partially redundant software may be justifiable. Git is one example of a highly successful package that was developed despite the existence of other VCSs because those tools failed to support highly distributed, branch-heavy workflows. Any such development, however, should be driven by a clear shortcoming of existing software.

Checklist for software integration

Once potential complete or partial solutions have been identified, they must be individually checked to see whether the following criteria can be satisfied:

- Ensure that the software is appropriately licensed to allow integration and modification.
- Ensure that the architecture and interfaces are designed for easy integration.
- Determine whether the software is actively maintained and whether code or documentation contributions would be accepted.
- Check whether the API is stable to estimate future maintenance effort.
- Confirm that all interfaces, the architecture and—in case modifications are required—the source code are sufficiently documented.
- Establish that the majority of the code base is properly tested and validated.

In many cases, software that does not satisfy these criteria can be brought into compliance with sufficient modification, either by the user (e.g. by writing documentation, adding tests, or maintaining a fork) or by contacting the maintainer (e.g. to add a license). If such modification ultimately proves insufficient to satisfy these criteria, however, the software should be discarded as a solution for the problem at hand. Note that all criteria outlined above extend to dependencies as well, so code bases with many dependencies must effectively clear a higher bar.

Guidelines for package adaptation

For packages that provide a near-complete solution but require some adaptation to completely address the relevant part of the problem, we have developed the following guidelines:

- Plan development: Unless it conflicts strongly with the protection of intellectual property, obtain feedback on the proposed modifications and extensions from the current package maintainer.
- Testing: Before modifying any code, implement any missing tests to ensure adequate coverage of code paths that may be modified.
- Validation: Add integration tests as needed to completely verify the correctness of the original software when applied to the problem at hand.
- Modularity: Refactor the code base such that all parts that require modification are mostly separated from those that do not.
- Adaptation: Write only as much code as is needed to address the problem at hand without aiming for too much generality.
Steps for refactoring prototypes into packages

If there are no applicable packages, but there are multiple existing prototypes, then these should be refactored into a package. Before refactoring, all related prototypes and packages should be analyzed for interfaces, overlaps, strengths and weaknesses, with a special focus on relevant packages that failed the tests of the checklist for software integration. Such software, which may have failed due to, e.g., licensing issues, likely contains a great deal of expertise and know-how that is invaluable for the design of any new package. It may also be worthwhile to seek additional information on these packages through public forums or issue trackers, or by reaching out to developers or experienced users for further insight.

The new package may then be implemented according to the following steps:

1. Determine the scope and the software architecture. Identify the core logic and then build layers around it for additional functionality such that dependencies only point inwards as described in the The Clean Architecture principle.
2. Draft the end user interface before implementation. Ensure that the most prominent use cases are supported by the drafted interface design.
3. Replace the prototype code with the package step by step within the original applications. Ensuring that the new code produces identical results at each step is an important part of the validation process.
4. Document all functions and interfaces well such that a user can understand and use the package without needing to consult any of the developers.
5. Write unit tests for all core functions and add integration tests for the main applications of the code.
6. After ensuring function and correctness, optimize critical code paths (and only those paths) if needed.

Guidelines for developing a new prototype solution

If new software is required, we impose the following minimal standards on the resulting prototype code to simplify its future refactoring if needed:

- **License**: Any code base should be licensed. Such a license may, for instance, explicitly permit internal reuse and require acknowledgment of the original author. To simplify matters, a research group could agree on a general license that is assumed to apply to all code that is not explicitly licensed otherwise. Organization-wide policies regarding intellectual property will apply.
- **Modularity**: Code should be developed as modularly as possible to simplify potential refactoring at a later stage. However, this modularity should not come at the expense of simple, problem specific interfaces that streamline solutions to the problem at hand.
- **Stability**: All code should be part of a version-controlled repository. Interfaces can be changed whenever convenient.
- **Documentation**: The code should have enough internal documentation to allow developers to modify it as well as sufficient API documentation for prospective users familiar with the code base.
- **Validation**: The code base should be validated as part of the research process, ideally against known results.
- **Testing**: Unit tests are not necessary unless they aid in the development process.

THE GLOTZER GROUP SOFTWARE STACK

We now trace the growth of the core software developed within our group. Although our initial development followed a far less structured pattern than lazy refactoring, the challenges we encountered and the experiences we accumulated informed the guidelines that we now follow. By describing our internal software ecosystem (illustrated in Figure 3), we aim to motivate our approach and provide a blueprint for how to use it to create a powerful, sustainable, and integrated software stack for domain-specific computational research.

For brevity, we restrict ourselves to software that is directly involved with the generation, organization and transformation of data. Therefore, we omit discussion of, for instance, all software used for producing illustrations or presentations. We also avoid discussing operating systems aside from noting that we primarily use UNIX-like systems such as various Linux distributions and Mac OS X.
Figure 3: We illustrate here the relationships between the various components of our software stack. Software packages are grouped according to their overarching functions. Interactions are denoted by adjacency. Software that is developed within our research group is shown in dark blue with white labels. Workflows for computational projects are typically organized using signac-flow, which links together simulation, analysis, and plotting. The vast majority of our simulation needs are provided by HOOMD-blue, but we occasionally utilize other tools, for instance to perform atomistic simulations of proteins. Which visualization toolkit is used depends on the specific simulation, and occasionally also on whether the toolkit has the required analysis capabilities built-in. More generally, the problem at hand dictates the appropriate analysis software packages. We use various plotting tools, especially Matplotlib, to visualize the evolution of both raw quantities (such as total system energy) as well as the outputs of more complex analyses. All data generated throughout this process, including raw simulation trajectories as well as the outputs from complex analyses, is stored and managed using signac.

Simulation

The genesis of our lazy refactoring approach lies in the consolidation of “single-use codes” written for various particle simulation techniques into a package called HOOMD-blue.11,12 HOOMD-blue evolved from HOOMD,11 a general purpose Molecular Dynamics (MD) program developed for bead-spring polymer models that was the first MD package to run completely on Graphics Processing Units (GPUs). HOOMD-blue exemplifies the paradigm of growth by refactoring and incorporation of external code. Today, HOOMD-blue runs many flavors of MD and MC simulations on both CPUs and GPUs, and it supports a rich variety of phenomena that were not originally envisioned. These features were all developed separately for specific research purposes, and the original implementations underwent many revisions before incorporation into a public release of HOOMD-blue.

Consolidating our development efforts has reduced development time, increased reproducibility, and improved code quality, allowing us to improve upon our original prototypes. Moreover, we have incorporated lessons learned from modifying preexisting toolkits that were not modular enough to allow easy modification or integration with our workflows. HOOMD-blue’s highly structured, object-oriented design maximizes ease of modification, enabling the continued integration of new tools such as the Hard Particle Monte Carlo (HPMC)13 module from a pre-existing code base.14
With internals written in C++ and CUDA, HOOMD-blue is one of the fastest particle simulation tools available: it is one of NVIDIA’s official benchmarks for new GPU hardware. Unlike toolkits such as Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS) or the GROningen Machine for Chemical Simulations (GROMACS), which use a specialized file format for simulation configuration, HOOMD-blue offers a full-featured Python interface for greater flexibility and ease of integration with other tools. An open-source toolkit, HOOMD-blue has benefited greatly from numerous contributions from its user base; at the time of writing, 29 developers external to the group have contributed to HOOMD-blue. HOOMD-blue also uses a number of external code bases, including: pybind11 for exporting C++ classes to Python; CUDA, cub and Thrust to utilize GPUs; the Message Passing Interface (MPI) to scale across multiple nodes; LLVM for Just-in-time (JIT) compilation; the Eigen linear algebra library; and cereal to serialize data for communication.

Data Analysis

For generic data analysis tasks, our group makes liberal usage of existing software, such as components of the SciPy ecosystem. In addition to NumPy, which we use extensively and have incorporated into many of our own open-source packages, we also use other SciPy tools for, e.g., optimization and working with spatial data. Jupyter notebooks are central to our workflow, and we typically use the Matplotlib library for plotting. For machine learning tasks, we typically develop methods using scikit-learn, Keras, and TensorFlow.

Visualization

Our simulation visualization codes have undergone the most consistent process of repeated refactoring of prototypes and partial solutions. Our simulations typically generate long trajectories of complex systems that require powerful software and hardware for effective visualization. HOOMD-blue provides output in formats that can be interpreted by well-known simulation visualization packages such as VMD, PyMol, and Ovito, but these tools are generally incapable of visualizing anisotropic particles, which require interpretation of their orientations in addition to their positions. Therefore, we used injavis, a previously developed Java visualization and analysis tool that represented particle data using XYZ-coordinates. Our group members extended this software to account for particle orientations, and we enabled HOOMD-blue to provide output data in this format. As a largely Graphical User Interface (GUI)-centric application, however, its analysis capabilities can be difficult to integrate into scripted workflows.

The goal of refactoring the functions in injavis into a more scriptable tool was a driving force for the development of freud. To leverage freud’s analysis capabilities for visualization, we developed a visualization toolkit called plato that outsources its analysis functions to freud. This division makes plato easier to adapt for new visualization tasks, resulting in a more modular, maintainable software infrastructure. In addition to plato, we developed fresnel to generate publication quality images using a GPU-accelerated path tracing engine.

Data and Workflow Management

A comprehensive simulation study typically involves conducting many simulations, often using HPC clusters to achieve meaningful system dimensions and time scales. Although HOOMD-blue scripts accommodate user-defined parameters, the user remains responsible for reliably associating these parameters with the simulation outputs and subsequent analyses. Many group members developed prototype solutions, but they suffered from numerous drawbacks with respect to scalability, flexibility, and interpretability.

To resolve these problems, we developed signac, an open-source framework for constructing complex workflows on large, heterogeneous data spaces. Inspired by these prototypes, signac uses well-formed JavaScript Object Notation (JSON) parameter files to associated data files with their identifying parameters, providing a robust and full-featured database interface to data stored directly on the filesystem. The system is
designed for the high performance filesystems inherent to HPC environments and supports the highly parallel file I/O operations required for, *e.g.*, MPI-enabled HOOMD-blue simulations. Since all data and metadata are stored directly on the filesystem, they can also be easily transferred using tools like ‘rsync’ or GLOBUS.24 The signac framework includes signac-flow, a tool for managing and automating complex workflows operating on a signac data space. Workflows designed with signac-flow are immediately portable to HPC environments, making it possible to design and test workflows on local resources and then immediately submit them to a cluster scheduler.

**Software Integration**

The smooth functioning of our overall pipeline also depends on a number of smaller packages that perform more limited but equally important tasks. A good example of applying lazy refactoring is rowan,25 an open-source Python package for quaternion operations. A standard method for representing particle orientations in 3D, quaternions are used throughout our code base; however, individual packages have historically each had their own implementations. This fragmentation strongly suggested the need for standardization, particularly because many individuals also reimplemented these methods for their own, more *ad hoc* uses. Although quaternion packages already existed within the Python ecosystem, they suffered from numerous drawbacks with respect to performance and flexibility that made them unsuitable for incorporation into our code bases. rowan is the result of refactoring our own quaternion codes into a single package providing a unified API for working with quaternions at a uniform level of generality appropriate for our needs.

**TRAINING AND SUPPORT**

Our group has built an infrastructure around our software to provide users with training and information on how to take maximal advantage of our ecosystem. In addition to the documentation associated with each of our packages, we have created comprehensive integrated documentation that explains how to utilize our stack as part of a cohesive workflow, including a crash course to incrementally acquaint new users with this stack. We use Slack to maintain internal chat rooms for instant technical support, specialized discussions of science or software, and coordinating development.

We also aim to make our software useful to the broader community. We publish documentation through ReadTheDocs, distribute software via Anaconda and PyPI, and have published papers on some of our packages to further publicize them. We have also presented our software at the Scientific Computing with Python (SciPy) conference,26 the annual American Physical Society (APS) March meeting, the triennial Foundations of Molecular Modeling and Simulation Conference (FOMMS), and the annual meeting of the American Institute of Chemical Engineers (AIChE), to name a few. In addition to these presentations, which are one avenue for feedback, we also use surveys and focus groups to gather user feedback. Bimannual hackathons where all group members participate in improving our code help to improve the distribution of code ownership.

**CONCLUSIONS**

Advances in computational science are heavily dependent on the ongoing evolution of the scientific software ecosystem. Although quickly produced *ad hoc* solutions may seem sufficiently expedient in the moment, scientific progress can often be accelerated by a judicious use of existing software. Consequently, time spent properly designing and developing reusable software solutions may prove worthwhile if the reuse potential is identified sufficiently early in the development process. Properly estimating the future reuse potential of a piece of code is a daunting task, however, requiring a degree of foresight that is often impossible in scientific applications. Moreover, assessing the existing software options when embarking on a new project is far from trivial.

To address the need for a systematic approach to these problems, we have outlined a near optimal approach to determining when and how to develop reusable software that works well for us. The lazy refactoring approach is designed to balance sustainably improving the scientific software landscape with making immediate scientific progress. It does this by advocating for individual researchers to

- evaluate existing software for its reuse potential prior to any code development,
- adapt existing code bases for the problem at hand,
- refactor existing code bases into proper packages whenever there are more than two use cases,
• develop rapidly evolving prototype code strictly focused on solving the problem at hand in all other cases.

Lazy refactoring is particularly well suited to the academic research group. To illustrate its application in this context, we have shown how it has been applied over time by our research group to develop a suite of independent but highly interoperable and powerful tools targeted at a specific class of scientific problems. Through this, we provided a concrete example of how lazy refactoring may be used to significantly accelerate net scientific output in any subfield of computational science.

ACKNOWLEDGMENTS

We would like to thank all contributors to the group’s code base, including all members of the Glotzer group as well as the numerous external contributors to our software listed on the respective websites. We would like to thank Michael Engel for his development of injavis, some of which was done while in the Glotzer group. The software toolkits, packages and methodologies described in this paper have been supported both internally (injavis and plato) and by a number of grants, principally: DOD/ASD(R&E) under Award No. N00244-09-1-0062, “Smart, Autonomous. Adaptive Phenomena in Self-Organizing, Reconfigurable Materials,” (2009-2014) for HOOMD-blue v0.7 – v1.1, Freund and later support for injavis; National Science Foundation, Award # DMR 1409620, “CDS&E: Fast, scalable GPU-enabled software for predictive materials design & discovery,” (2014-2018) for HOOMD-blue development, especially DEM and HPMC, and for fresnel, as well as project conceptualization and early implementation for signac; and MICCoM, as part of the Computational Materials Sciences Program funded by the U.S. Department of Energy, Office of Science, Basic Energy Sciences, Materials Sciences and Engineering Division, under Subcontract No. 6F-30844 for development and deployment of signac. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the DOD/ASD(R&E).

REFERENCES