Developing Community Codes for Materials Modeling

Steven J. Plimpton
Sandia National Laboratories
Albuquerque, NM

Julian D. Gale
Nanochemistry Research Institute, Department of Chemistry
Curtin University
PO Box U1987
Perth, WA 6845
Australia

sjplimp@sandia.gov, J.Gale@curtin.edu.au

Keywords: materials modeling, open source software,
molecular dynamics, lattice dynamics, force fields

Abstract

For this article, we call scientific software a *community* code if it is freely available, written by a team of developers who welcome user input, and has attracted users beyond the developers. There are obviously many such materials modeling codes. The authors have been part of such efforts for many years in the field of atomistic simulation, specifically for two community codes, the LAMMPS and GULP packages for molecular dynamics and lattice dynamics respectively. Here we highlight lessons we have learned about how to create such codes and the pros and cons of being part of a community effort. Many of our experiences are similar, but we also have some differences of opinion (like modeling vs modelling). Our hope is that readers will find these lessons useful as they design, implement, and distribute their own materials modelling software for others to use.

1 Introduction

As described in the abstract, community codes are openly-available software created and maintained by a (small or large) team of developers for a broader user community. The benefits of such efforts for users are obvious; they have free access to software that is continuously improving, resources to turn to for help, as well as input to the development process by requesting bug fixes or suggesting new features. Our experience has been that sponsoring a community code is also a net positive for the developers. Aside from the attendant glory and wealth associated with free software (ok, that's a stretch) the benefits chiefly result from your code having users. Specifically,

- users find and report bugs
- users suggest improvements and new capabilities for the code
- users may implement new ideas themselves and give you new or improved code
- users cite your papers
- users maybe become new colleagues, leading to new collaborations

The only downside of having users is the extra work required to support them. This effort comes in the form of documenting your code's capabilities, answering questions, and responding to user feedback. The level of support offered is up to you, so it does not have to be onerous. After all, users get what they pay for, and the code is free. Indeed many such codes come with express statements that support cannot be provided. However, in reality most developers relent and are willing to offer as much assistance as time permits. As a user community enlarges, support can be offered by a growing group of people via web-based forums and mail lists.

As examples of community codes for materials modeling, we highlight our experience with LAMMPS and GULP, since the authors are their lead developers. In order to give context to the subsequent discussion, we start with a short historical perspective on these two codes.

LAMMPS is a classical molecular dynamics (MD) code [13,15], begun in the mid-1990s as a cooperative effort between two US DOE laboratories and several industrial partners to develop a parallel MD code, since parallel machines were then an up-and-coming novelty. For its first 10 years LAMMPS was free but required new users to sign a perfunctory license agreement, which about 100 users did. In hindsight such a license was a significant barrier to attracting users since it often meant a lawyer wanted to read it. In 2004, we re-wrote the code in C++, to make it more flexible for adding new features, and released it openly under the GNU General Public License (GPL) [12]. It was downloaded 1000 times in the first four months, and 150,000 times to date. Since its open release LAMMPS has grown in size from about 50K to 500K lines of code, as developers and 100+ users have contributed new code. Some of the new capabilities are features we never imagined being part of LAMMPS or even an atomistic MD code, such as treatment of electrons as variable-radius particles [11], continuum-scale models of fracture dynamics via particles [14], or coupling granular particles to finite-element fluid solvers for two-phase flow modeling [2]. Our chief mechanism for supporting LAMMPS users is a mail list where questions or problems can be posted, which we began a year after the code's open release. The list now has 1200 subscribers and an archive of 40K postings.

Similar to LAMMPS, GULP is also based on exploiting a classical force field description of interatomic forces [5, 7, 8], though there was also a brief flirtation with periodic semi-empirical quantum mechanics [6]. In contrast to LAMMPS, GULP targets the niche application of lattice dynamics, though it also has a molecular dynamics capability. Because of this focus, our objective has been to provide high levels of analytical derivatives that allow accurate calculation of mechanical and phonon properties for crystalline materials. GULP also started in the early 1990s, at the Royal Institution of Great Britain, as an attempt to automate the fitting of interatomic potential parameters [4], which previously had to be undertaken by hacking source code for each desired fit. At that time disk space was at a premium and was often exhausted by building a different executable for every job!

Initial distribution of GULP began by sharing copies with other groups in the UK, but eventually progressed to the point where it was distributed to any academic group by emailing tar files. In the late 1990s, this was placed on a more formal footing by Imperial College. Instead of opting for an open-source approach, a different licensing arrangement was made. While the right to free access for academics was enshrined in the agreement, commercial distribution was assigned to the company that is now Accelrys Inc. In this respect, and the fact it remains a staunchly proud Fortran code, the pathway for GULP has forked considerably from that taken by LAMMPS. However, there are also many similarities, including the size of the current version which also runs to nearly 500K lines of code. Today, academic distribution is handled by an automatic web-based registration system; access for anyone with a University email address is thus almost instantaneous, similar to a GPL code. Because of the different access mechanism, we tally registered users rather than downloads; this currently runs to more than 6,000 people.

The next section distills six lessons we have learned about what helps a materials modeling code attract a community of users. Many of the ideas reflect the current state of LAMMPS and GULP, but we learned them by trial-and-error and implemented them incrementally. We argue that turning your home-grown research code into a community code is not a decision with a large energy barrier, but is more a philosophical approach to software design, development, and release. Like most software tasks, adopting a community-oriented strategy is least difficult if it is part of the up-front design of your software and maintained incrementally over time. But of course existing legacy software can also be released openly and become a community code; hopefully these ideas will benefit that process as well.

2 Lessons learned

Here are six rules-of-thumb for creating and maintaining a successful community materials modeling code:

1. make something people want
2. the perfect is the enemy of the good
3. make it easy for others to understand, modify, and extend your code
4. choose an appropriate license
5. support your users
6. choose an appropriate name for your software

2.1 Make something people want

This is a mantra of the Internet start-up culture [9] when a handful of friends create a company to turn their software idea into money. In a commercial setting, it seems obvious that customers will only visit your web site, use your software, and give you money if they get some value from it. But it also applies to freely-available research software. Knowing how to write code to perform some computational task is a necessary first step, but is not sufficient to attract users to your software.

Asking yourself questions such as these is a useful exercise: What problem do people want to solve? How can our software make it easier for them to do so? In what ways will our code be different or better than others that already exist? It also helps to examine your software from a new user's perspective. Every programmer knows it's easy and fun for an expert (you) to write code with only yourself in mind, adding features or obscure and tricky options that confuse mere mortals. If your software is not easy for a new user to quickly do something useful with, they probably won't continue to use it. If your code has a mailing list where new users post questions, you will be surprised by what issues they stumble over which you never thought would be important.

The above points are illustrated by our initial motivation for creating GULP. At the time, there had been a series of codes going back several decades that already fulfilled the same basic need to optimize the structure of solids from interatomic potentials and compute their properties. This is nicely captured in a tribute to the pioneering contribution of Michael Norgett [19]. So why create yet another program in this field? Largely, this came from frustration, as a user of the programs of the day, that the input file format was too rigid and less friendly than it could be. Furthermore, for those not of a C persuasion, the timing coincided with the arrival of Fortran 90 and dynamic memory allocation. By removing the need to regularly recompile for each problem, this allowed distribution of executables for the benefit of those not inclined to worry about the finer details of computers. Thus experience as a user, combined with changes in technology, created the opportunity for a new code.

2.2 The perfect is the enemy of the good

This aphorism is attributed to Voltaire, who apparently was a savvy software developer. When you first contemplate releasing your code to the unwashed masses, you imagine users will pore over its innards, test every option, and mock you whenever it breaks. Thus the natural tendency is to wait to release until you are confident the code is near-perfect. Aside from the improbability of ever reaching that state, the problem with this strategy is you delay having users, with all the benefits listed in the introduction.

A better mantra for community codes is release early and often. For example if your current code has 5 simple bullet-proof features and 3 bleeding-edge brittle ones that are only suitable for experts, you are better off removing the 3 features and making an initial release of the simpler version of the code. You may get feedback that a capability you hadn't thought of is more useful than the 3 you are working on. Or when you do release the bleeding-edge features (one at a time), you will hopefully have users eager to try them out and give you feedback about what works and what doesn't and what would make the new feature easier to use. Having such beta testers has a synergistic effect on the development process, speeding the rate at which a bleeding-edge feature is converted into a bullet-proof one. It's also more satisfying to a developer to release something immediately and get feedback than it is to wait for perfection.

With LAMMPS, we initially released a new version of the code a few times a year. The releases became artificial deadlines which developers stressed over getting code ready for. Instead, we now release every bug fix or new feature as soon as we finish it, posting a patch and new tarball on our web site, often 100s of times per year. If the patch isn't 100% correct or breaks some other portion of the code we typically hear about it from users, more quickly than if we delayed release and tried to test it rigorously ourselves. Our sense is that users also appreciate having immediate access to a bug fix or new feature, rather than waiting for a periodic release. Likewise, users who contribute a new feature like the reward of seeing their code immediately added to the public distribution. A happy by-product of this strategy is that we only support one version of the code, the most current.

In the case of GULP, the different distribution mechanism has led to a hybrid model that has some merits and some downsides relative to the LAMMPS approach. First, consider the negative side. While updates to GULP was initially released with great regularity in its early days, in recent times the releases are much less frequent than can be achieved with open source. That said, bug fixes are quickly made for academic users via the web site. The main difference with having a commercial distributor is that it drives a tendency to focus on completing new features for an annual cycle. On the positive side, commerical software companies have a stringent quality assurance process that is just as valuable as user input in ensuring robustness of code. This often includes throwing what the developer might consider to be unphysical examples at the code, in an attempt to break it. As a result, it becomes possible to provide warning messages, improve documentation with physical insights as to reasonable valuables, and to check for input parameters that might cause the user grief.

We also note one caveat to the suggestion to release regularly to users, rather than waiting for unobtainable perfection. Don't release new options that aren't fully documented (which may include warnings and usage advice). It's important that both old and new users can expect the code to work as its documentation advertises.

2.3 Make it easy for others to understand, modify, and extend your code

There is zero chance your software will do everything many users want it to. Some will want a one-off feature needed only for their current problem. Others will have ideas for more generally useful features. If you make it easy for users to add such capabilities, they will be more likely to use your code, and more likely to contribute code they write, for you and others to take advantage of.

The first step to making a code easy to modify, is making it easy to understand. This starts with coding style. Having a consistent and readable format for your source files, with comments describing data structures and code operations is quite helpful. Opting to implement tricky algorithms in simple, straight-forward manners, rather than clever and opaque is also a good rule-of-thumb. Having a developer's guide that outlines the basic structure of the code is also useful when programmers look at your code for the first time to get the big picture of how it works. All of these things are just as invaluable to the core developers, since few of us have the memory capacity to remember why we coded something in a particular way years later unless the code or documentation explains it.

Though LAMMPS is written in C++, we opted not to take full advantage of C++ complexity (e.g. operator overloading, some forms of templating), partly because we felt it can make code harder to understand for casual readers. Instead LAMMPS is designed in more of an object-oriented C style where low-level data structures and performance-critical kernels are written in simple C-like syntax. We leverage C++ and its classes mostly at a high-level in the code to provide flexibility and extensibility as discussed below. We note that this more limited form of object-orientation can be used in many languages, including Fortran. A benefit of this approach has been that when a user has a question about how some interatomic force is computed or a thermostat works, they can often find, read, and understand the pertinent lines of code with minimal effort. This means bugs are found more quickly and users gain confidence they understand how the code operates.

Additional steps to make it easy for users to extend your code are to minimize the amount of existing code they need to understand and interact with, as well as the amount of new code they need to write. Streamlining this process was one of our chief goals with LAMMPS. This was based on our own experience that when modeling a new system or phenomenon with classical MD, we often needed to add a new interatomic potential, or new boundary condition, or new diagnostic capability to the code.

LAMMPS tries to achieve easy extensibility by "styles", which are enabled by object-oriented concepts. The core of LAMMPS includes several parent classes which define an interface for classical MD constructs like potentials (a *pair* style) and diagnostic computations performed as a simulation runs (a *compute* style). The most general is something we call a *fix* style, which allows specific computations to be performed at specific points within the timestep loop, e.g. enforcement of a boundary condition or addition of a force constraint. The core of LAMMPS is small; over 90% of the code base is add-on child classes which each implement a specific instance of a style, e.g. a Lennard-Jones or REBO potential, or a *compute* that calculates a pressure or mean-squared displacement for diffusing atoms. Since the child class inherits the interface definition from the parent class, creating a new child class only requires writing code for a handful of methods that implement the interface.

We have found the above approach to have several benefits for users and developers alike. First, a user only needs to understand the interface for the style they are adding. The mechanics of adding the feature are simple: write the child class, put the *.cpp and its *.h file in the source directory, and re-compile the code. Since the new class also defines the syntax (style name and arguments) for how the feature is invoked from an input script, it can then be tested and used immediately. Because the rest of LAMMPS knows nothing about the child class, but only the interface defined by the parent class, all capabilities of the existing code can effectively use the new feature as a black box. This also makes it easier for the developers to incorporate user contributions into the master version of the code. So long as users avoid making changes to the core of LAMMPS, we can add their new style classes to the public release quickly, with minimal inspection or concern that it will break other LAMMPS functionality.

Another way users will want to "extend" your code is to use it in tandem with other codes. This may be as simple as using another code to pre- or post-process data that is input/output by your code. If your code works with common data file formats, or you provide auxiliary tools that translate between your data format and others, or you at least carefully document the input and output formats used by your code, it will be easier to use your code as part of a multi-stage workflow.

Users may also want to use your code with other codes in a more coupled manner, e.g. to model a multi-physics or multi-scale problem. This could be a loose coupling where an umbrella code invokes your code for a few timesteps, passes boundary data to a second code, invokes it, passes data back to your code, and iterates. Or it could be a tight coupling where at every timestep a multi-scale simulation requires both codes to perform calculations, e.g. to contribute rows to a coupled matrix which is then inverted.

Your code will more easily allow for these usage modes if it is not simply a stand-alone executable, but can also be built as a library, so that another code can make function calls to it. This can often be accomplished by a bit of reorganization at the highest level of your code. As an example, LAMMPS is fundamentally a library. To build it as a stand-alone code, a small main.cpp file is added with a main() function which simply creates an instance of LAMMPS, followed by a function call to process the command-line input script. Other library calls are provided that allow an external program to invoke input script commands one at a time, or retrieve or reset internal LAMMPS settings or data values such as atom coordinates. Users can also add their own custom library interface functions.

We chose a generic C-style API for the library interface, which means calls can be made from virtually any programming language, including C, C++, Fortran, and scripting languages like Python. The scripting capability is particularly powerful, as it allows users or companies to quickly create infrastructure around your code, e.g. to write a GUI that drives it or connect it to plotting or analysis or visualization packages that exchange data with it.

Other good programming practices which insure your code (library) interacts nicely with other software are to use no global variables, encapsulate the code in its own namespace, and to insure it can be instantiated multiple times. For example, the latter is useful if a continuum model wants to create multiple regions of fine-scale particles, and run each of them as an independent MD simulation.

We don't claim that GULP was as well-planned and structured as described above for LAMMPS. Rather, GULP has experienced the typical problems of a code written with the dual purpose of being a vehicle for the developer to learn about programming. This has been compounded by the evolving state of the Fortran language. Early versions inherited what are now deemed "bad practices" of Fortran77 codes, such as goto statements and common blocks. Therefore GULP has had to be re-written several times over its lifetime, partly to migrate to Fortran90, but more importantly because the vision for what the code might do has changed. Certainly the most valuable lesson that has come from this, is that it is important to have the broadest possible horizon for where your code might eventually go when planning its structure, even if the implementation may be many years in the future. Modularity and transparency in programming are also vital. When creating new code we have the best practice we have found is to first write all the comments that define what will happen and in what logical sequence; the rest is then easier as it's a matter of filling in the gaps with detailed code.

A final suggestion is to devise ways to recognize users who contribute to your code, i.e. to make them feel part of your code's community. For LAMMPS, we list their name at the top of source-code files they contribute and on an "authors" page on the web site. If a contributor has a published paper describing the functionality they added to LAMMPS, the code also outputs the paper citation when a user invokes that functionality. Similarly for GULP, contributors are acknowledged at the start of the manual and on the main web page, as well as being documented in the source code. The hope is that these mechanisms will increase the likelihood of users citing the contributor's work.

2.4 Choose an appropriate license

A license agreement typically spells out rules for how people can use your code or further distribute it, as well as (the lack of) liability you assume for its use. In a legal context, whoever contributes to a community code becomes an author who owns the copyright to pieces of code they contribute. Thus they have the right to determine how their portion of the code is licensed for others to use.

As mentioned in the introduction, LAMMPS is an open-source code, meaning it is distributed freely to anyone with an accompanying open-source license. When we first thought about releasing LAMMPS in this manner, we worried about making it openly available on a web site. We imagined people might "steal" it, change the variable names, and claim it as their own. Or they might harvest it for brilliant ideas and scoop us on future papers. Now in hindsight (or possibly just cynicism and old age) we realize brilliant ideas are few and far between and are rarely encoded in software. And we now believe the benefits that accrue from open source and more users (see the introduction) far outweigh those irrational fears.

There are many variants of open-source licenses [10]; here we list three basic kinds and what we consider their key attribute from a scientific software perspective. In all cases, any user is free to use and modify your software as they wish. What the licenses prescribe is how a modified version must be licensed, if the user chooses to re-distribute the modified software to others.

- GPL = GNU General Public License: both changes to your software, and new software that links to your software, must be distributed under the GPL
- LGPL = GNU Lesser General Public License: only changes made to your software must be distributed under the LGPL
- BSD = Berkeley Software Distribution: imposes no restriction on how modified software is distributed

If your code is licensed under the GPL, it means anyone who enhances it (and distributes their changes) must also do so under the GPL, i.e. make their source-code modifications openly available. This is not true for new software they write which simply invokes your code as an executable, but it is true for new software that interfaces directly with your code, e.g. calls it as a library. This is often viewed by companies as a dis-incentive to using your code, particularly software companies, since new code they write and wish to sell, which interfaces with your code, must also be open-sourced.

The LGPL is less restrictive; only changes made to your code remain open-source. A company can create proprietary code that wraps or otherwise interfaces with your code through its library interface, and not be required to distribute it in an open-source manner.

The BSD is least restrictive of all; another party is free to embed your code or parts of it in a proprietary product which they sell as closed-source software.

LAMMPS is currently a GPL code, because we didn't initially think of its utility as a library. If we had the chance to do things over again, we would choose LPGL as a good compromise license which allows companies to add value (proprietary software) around a community code while protecting their investment. At the same time, changes they make to your code to enable or optimize its use in that mode, remain open for everyone to benefit from.

While it may seem premature to worry about these issues before a community code becomes popular, it's actually easier to decide on a license up-front rather than later. This is so all the authors of your code agree on how their contribution is being distributed. You can always change the way your code is licensed later (the authors own the copyright), but it does require buy-in from each author in order to include their contribution in a newly-licensed version.

As highlighted in the introduction, GULP has taken a different pathway to that of open source, opting for a hybrid model that allows free distribution to academics, but also providing the opportunity for commercialization. In hindsight this came about by gradual evolution rather than as a considered decision. At the time GULP was transitioning from an in-house code to a community code, the norm was to "protect" source code, rather than make it open. Once a code is locked into a commercial license agreement it is then it is much harder to change course, so it is worth taking the time to seriously consider the right distribution model for your software upfront. Obviously there are benefits to commercialization - just don't expect to get rich quickly (or at all!). Having some income from a community code does make life easier in academia, where funding agencies and employers rarely cover the full costs of research. It can help relieve the pressure of supporting the operating costs of a group, thereby giving you more time for software development. However, the major downside is that a commercial license agreement can be a barrier to rapid growth of a code, since incorporating contributed developments is a more complex issue, and users are less likely to donate new features. Given these constraints, GULP has remained largely a single-developer code, though there have been contributions by others to capabilities such as genetic algorithms [20], and neutron scattering [3, 16].

In the case of force field lattice dynamics, which is a relatively mature field, the model of a very small and restricted development team can work. However, one of the authors (JDG) is also involved in the development of a density functional theory-based program for quantum mechanical studies of complex materials, namely SIESTA [1, 18]. In this field it would be nearly impossible to keep up with the fast-moving research and the rapid evolution of high performance computers without a much larger development team. However, a revenue stream is also important to provide for code enhancement and user support. For SIESTA, the commercial dimension was managed through creation of a non-profit charitable foundation so that royalties can be plowed directly back into the code.

Based on the above, it is our view that open source is almost certainly the model that will dominate the future of large, rapidly evolving scientific software packages. However, the nature of commercial scientific software is changing as well. Some companies are shifting from developing core codes in house, to facilitating the use of externally-developed codes through graphical interfaces, productivity tools, and the provision of guaranteed support. Hence the future may offer new ways to both be open and to provide revenue to support on-going code development.

2.5 Support your users

Think about what frustrates you when you use someone else's code, including commercial software. If its features aren't well documented, if it doesn't do what you expect and you can't figure out why, if you don't know how to interpret or fix errors when they occur, if you're not sure it's performing the calculation you want, or most importantly, if you have no one to turn to when you have a question, then you are less likely to continue to use it and trust the results. These are the kinds of issues you want to address for your community code to increase its scientific utility and the productivity of your users.

For LAMMPS, we've taken a three-pronged approach to user support which has morphed over time as our user community has grown: an on-line manual, a web site, and a mail list. Here are some details of how we use and maintain each.

The LAMMPS manual is meant to be a comprehensive resource for users that documents all the features the code supports and the meaning of error messages they might encounter. It's also the only way the developers can remember what the code is supposed to do. We enforce this by requiring ourselves and others who contribute code for a new *style* (see section 2.3) to also provide a new documentation page before we will release it. Speaking from experience, even if your community code is now tiny and as-yet unreleased, forcing yourself to write a basic manual describing the code's current capabilities, then adding to it incrementally, is much less painful than waiting to give birth to a fully-grown manual in the distant future.

The LAMMPS web site is also oriented towards user support. Basically we post any content we generate or that users contribute which we think could benefit other users. This includes a searchable archive of the mail list, example scripts and animations of simple simulations included in the LAMMPS distribution, vignettes and images/animations from user simulations which they contribute, citations and abstracts for publications that reference LAMMPS, slides from tutorials or workshops the developers have presented, and links to 3rd-party tools and web sites that LAMMPS users could find useful.

The mechanism for maintaining both the manual and the web site is as simple and automated as we can make it. We edit or auto-generate text files on our own machines which are auto-converted to HTML and pushed to the web server. The focus is on content, not eye-catching HTML, since we are novices at web site design. In recent months, users have contributed pre-built LAMMPS executables for OS X, Windows, and some Linux distributions, which new users find helpful if they want to avoid learning how to build the code themselves. Likewise, some remote developers host mirrored SVN and Git repositories so that users can more easily stay current with code updates.

The LAMMPS mail list is meant to be a community resource where new or experienced users can post how-to questions or details of problems they are having and get timely answers. While answering questions has been a time-sink for the core developers, it has become less so over time as more users contribute answers and become recognized experts in facets of the code they either wrote or understand well. The manual and web site are also resources for answering questions; sometimes we can simply point the questioner to a relevant link.

We note that there are at least two powerful, free resources for hosting a community code on-line: Source-Forge and Google Code/Groups. Both provide download capabilities, mail lists, and web-site hosting, as well as other user- and developer-oriented options which you may find useful, such as forums, wikis, and bug-tracking systems.

While GULP has a different licensing arrangement, there are many similarities in the philosophy towards documentation and, to a lesser extent, support. For instance, all new keywords and options, which represent the two types of input/control mechanisms for the code, are documented in a file help.txt that is distributed with the code. This is automatically then parsed into html and uploaded as a web page. The same information is listed in the manual, which also provides an overview of the underlying theory, as well as the practical philosophy behind the software.

As with many codes, example files are also provided with GULP, to illustrate the use of features. These also serve as a test suite for quality assurance. A good philosophy is to provide an example for every new feature, as well as a sample output (note that this is an aspirational goal for GULP that is currently being implemented!). Automating the running of examples and comparison against previous output is a valuable tool for checking that source code changes haven't had unintended consequences elsewhere in the program. This validation process has to be account for differences in precision due to machines and compilers.

Finally, in terms of support for GULP there is a major difference from an open source code. Users with a commercial license have an assurance that they will receive a rapid response to their queries, and the availability of a graphical interface specifically designed for the package removes many of the input syntax or data post-processing issues that otherwise arise. Of course, in academia it is often not possible to afford the luxury of paying for support, so there are other options too. First, there is a frequently answered questions (FAQ) on the GULP web site to document the most common sources of puzzlement. Second, there is an email address to ask for (limited) help from the developer. Of course this comes without guarantees as the developer has a day job. Third, there is a user's forum at *http://www.gulpforum.com* which is run independently of the developers so that the community can share tips and solutions to problems.

2.6 Choose an appropriate name for your software

Although somewhat more trivial than the other lessons, the question of choosing a suitable name for your code is worth some consideration, as you might be stuck with the consequences for many years to come. To quote Shakespeare, "What's in a name? That which we call a rose by any other name would smell as sweet." [17] While there is definitely truth in this statement, having a memorable name for your software can be beneficial. Furthermore, in the current era having a name that is easy to find on the internet is also important. Somewhat fortuitously we named our package LAMMPS and not LAMPS. Otherwise searches would prove frustrating (yet illuminating). In the global world of science, it is also worth road-testing any name for cross-cultural sensitivities and meanings in other languages. In this sense, GULP may not have been an ideal name, since it has been reported that there are colloquial meanings in other European languages. Fortunately, none of these are sufficiently negative to cause the software name to be truly embarrassing when users mention it in presentations.

3 Conclusions

In this brief paper we've presented lessons learned about sponsoring community materials modeling codes, based on our experiences with LAMMPS and GULP. Our hope is that the ideas will be useful to others thinking about releasing their codes as tools for the materials modeling community to use. We hope that you, like us, are convinced that openly sharing your algorithms, software, and simulation tools in a collaborative manner is a great mechanism for helping science advance more rapidly.

A final cautionary lesson to share is the importance of managing expectations, both yours and whoever your bosses are:

First, in the spirit of *caveat emptor*, we offer no guarantees that the ideas presented here are sufficient to assure success of a community code. To wit, one of the authors (SJP) has released several other codes following these guidelines that have attracted nearly no users. If you do choose to distribute your software then also remember *caveat venditor*, and don't oversell your product or your resources.

Second, our experience has been that there are few institutional rewards to sponsoring or even working on a community code. The management you answer to typically cares more that your code models something useful and attracts funding, and less about whether you share it with others. Likewise, funding agencies do not often give grants for writing documentation, supporting users, or maintaining a web site. Indeed in some countries the development of software is expressly forbidden with government grant funding. Of course, if your software is successful and you document it in a manuscript that becomes highly cited then your institution might forgive you for supporting a community code.

So aside from the advantages listed in the introduction that accrue from having users for your code, the benefits of being involved in a community code are less tangible. For us, the bottom line is that if you derive satisfaction from helping create a software tool that others enjoy using and can do good science with, then being part a community code effort is worth the additional work.

Looking ahead, we are currently at the point where a significant shift in scientific computing hardware is occurring. The days of guaranteed Moore's law increases in performance due to decreasing feature size are at an end. Instead there is now a plethora of changing architectures with heterogeneous nodes, such as hybrid CPU/GPUs or many-core chips. These new paradigms pose new challenges to software developers and existing codes. A positive side-effect is that this will create new opportunities for the next generation of materials modeling codes.

4 Acknowledgments

The ideas in this paper were originally discussed in presentations by the authors at the 2012 Sanibel Symposium. We thank Simon Phillpot for organizing that session of the symposium and inviting us to speak.

SJP works at Sandia National Laboratories which is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. JDG acknowledges the support of the Australian Research Council through a Discovery Project.

References

[1] E. Artacho, E. Anglada, O. Dieguez, J.D. Gale, A. Garcia, J. Junquera, R.M. Martin, P. Ordejon, J.M. Pruneda, D. Sanchez-Portal, and J.M. Soler. The SIESTA method; developments and applicability. *J Phys-Condens Mat*, 20:064208, 2008.

[2] LIGGGHTS Open Source Discrete Element Method Particle Simulation Code. Web site: `http://www.liggghts.com/www.cfdem.com`.

[3] E.R. Cope and M.T. Dove. Pair distribution functions calculated from interatomic potential models using the general utility lattice program. *J. Appl. Crystallogr.*, 40:589–594, 2007.

[4] J.D. Gale. Empirical potential derivation for ionic materials. *Phil. Mag. B*, 73:3–19, 1996.

[5] J.D. Gale. GULP: A computer program for the symmetry-adapted simulation of solids. *J. Chem. Soc., Faraday Trans.*, 1:629–637, 1997.

[6] J.D. Gale. Semi-empirical methods as a tool in solid state chemistry. *Faraday Discuss.*, 106:219–232, 1997.

[7] J.D. Gale. GULP: Capabilities and prospects. *Z. Krist.*, 220:552–554, 2005.

[8] J.D. Gale and A.L. Rohl. The General Utility Lattice Program (GULP). *Mol. Simul.*, 29:291–341, 2003.

[9] P. Graham. Web site: `http://paulgraham.com`, see his essays entitled ”Be Good” and ”How to Start a Startup”.

[10] Open Source Initiative. Web site: `http://opensource.org/licenses`.

[11] A. Jaramillo-Botero, J. Su, A. Qi, and W. A. Goddard III. Large-scale, long-term nonadiabatic electron molecular dynamics for describing material properties and phenomena in extreme environments. *J. Comp. Chem.*, 32:497–512, 2011.

[12] GNU General Public License. Web site: `http://www.gnu.org/licenses/gpl.html`.

[13] LAMMPS molecular dynamics package. Web site: `http://lammps.sandia.gov`.

[14] M. L. Parks, R. B. Lehoucq, S. J. Plimpton, and S. A. Silling. Implementing peridynamics within a molecular dynamics code. *Comp. Phys. Comm.*, 179:777–783, 2008.

[15] S. J. Plimpton. Fast parallel algorithms for short-range molecular dynamics. 117:1–19, 1995.

[16] D.L. Roach, D.K. Ross, J.D. Gale, and J.W. Taylor. The interpretation of polycrystalline coherent inelastic neutron scattering from aluminum. *J. Appl. Crystallogr.*, 46:in press, 2013.

[17] W. Shakespeare. Romeo and Juliet. *J of Famous Plays*, 1:1, 1597.

[18] J.M. Soler, E. Artacho, J.D. Gale, A. Garca, J. Junquera, P. Ordejn, and D. Snchez-Portal. The SIESTA method for ab initio order-n materials simulation. *J. Phys. Cond. Matter*, 14:2745–2779, 2002.

[19] A.M. Stoneham, C.R.A. Catlow, and A.B. Lidiard. Editorial: Michael norgett (1943-2003). *J. Phys. Cond. Matter*, 16, 2004.

[20] S.M. Woodley, P.D. Battle, J.D. Gale, and C.R.A. Catlow. The prediction of inorganic crystal structures using a genetic algorithm and energy minimisation. *Phys. Chem. Chem. Phys.*, 1:2535–2542, 1999.