State of the Practice for Lattice Boltzmann Method Software

Spencer Smith[a], Peter Michalski[a], Jacques Carette[a], Zahra Motamed[b]

[a]McMaster University, Computing and Software Department, 1280 Main Street West, Hamilton, L8S 4K1, Ontario, Canada
[b]McMaster University, Mechanical Engineering, 1280 Main Street West, Hamilton, L8S 4K1, Ontario, Canada

Abstract

We analyze the state of software development practice for Lattice Boltzmann solvers and find by quantitatively and qualitatively measuring and comparing 24 software packages for 10 software qualities (installability, correctness/verifiability, reliability, robustness, usability, maintainability, reusability, understandability, visibility/transparency and reproducibility). Our reproducible analysis method employs a measurement template (containing 108 measures that are manually and automatically extracted from the project repository) and developer interviews (consisting of a set of 20 questions). From the measurement results, we ranked the software using the Analytic Hierarchy Process (AHP). Our ranking was roughly consistent with ranking the repositories by the number of GitHub stars, although the number one project by star count (Sailfish) is ranked 9th by our methodology. Our finding is that the state of the practice is healthy with 67% of the measured packages ranking in the top 5 for at least one quality, the majority of LBM generated artifacts corresponding to general recommendations from research software developers, common utilization (67% of packages) of version control and adoption of a quasi-agile development process.Areas of best practice to potentially improve include adoption of continuous integration, API documentation and enforcement of programming style guides. We interviewed 4 developers to gain insight into their current pain points. Identified challenges include lack of development time, lack of funding, and difficulty with ensuring correctness. Developers are addressing these pain points by designing for change, circumventing the oracle problem and prioritizing documentation and usability. For future the best practices of the future, we suggest the community consider: employing linters, conducing rigorous peer reviews, writing and submitting more papers on software, growing the number of contributors and augmenting the theory manuals to include more requirements relevant information.

Keywords: Lattice Boltzmann Method (LBM), research software, software engineering, software quality, Analytic Hierarchy Process

1. Introduction

We analyze the development of Computational Fluid Dynamics (CFD) software packages that use the Lattice Boltzmann Method (LBM). LBM packages form a family of algorithms for simulating single-phase and multiphase fluid flows, often incorporating additional physical complexities (Chen and Doolen, 1998), such as reflective and non-reflective boundaries. LBM considers the behaviour of a collection of particles as a single unit at the mesoscopic scale, which lies between the nanoscopic and microscopic scales. LBM solvers predict the positional probability of a collection of particles moving through a lattice structure following a two step process: i) streaming, where the particles move along the lattice via links; and, ii) colliding, where energy and momentum is transferred among particles that collide (Bao and Meskas, 2011). As an example of the output of LBM, Figure 1 presents the streamlines of converged flow past a stationary circular cylinder with varying Reynolds numbers. [Zahra can likely provide a better example. — SS] LBM has several advantages over conventional CFD methods, including a simple calculation procedure, improved parallelization, and robust handling of complex geometries (Ganji and Kachapi, 2015).

A small sample of important applications of LBM include the following: designing fuel cells (Zhang et al., 2018), modelling groundwater flow (Anwar and Sukop, 2009), and analyzing the flow of blood in the cardiovascular system (Sadeghi et al., 2020). As these examples illustrate, LBM can be used for tackling problems that impact such areas as environmental policy, manufacturing, and health and safety. Given the important applications of LBM, users of the libraries will be concerned with software qualities like reliability, robustness, reproducibility, performance, correctness

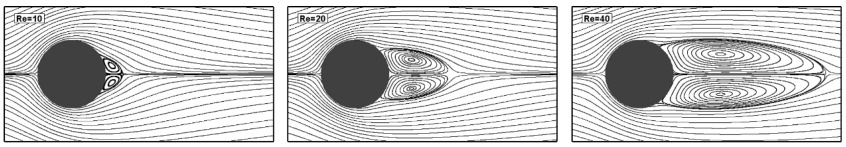

Figure 1: Streamlines of flow past a stationary circular cylinder at Reynolds number = 10, 20, and 40 (Chen et al., 2021)

and verifiability. Since their time is valuable, developers that create or modify LBM libraries will have additional quality concerns, including maintainability, reusability, and understandability. With the quality considerations in mind, and the potentially overwhelming number of choices for existing LBM libraries, the time is right to assess the state of the practice. The goal of this report is to analyze the current state of LBM software development and compare it to the development practices employed for research software in general. We want to highlight success stories that LBM developers can share amongst their community, and with the broader research software community, while at the same time looking for areas for potential future improvement.

To focus our efforts, we devised 10 research questions. The questions are used to structure the discussion in the paper, so for each research question below we point to the section that contains our answer. The questions are inspired by the research questions that underpin our data collection methodology (Smith et al., 2021). We start with identifying the examples of LBM software where we have access to the source code:

RQ1: What LBM software projects exist, with the constraint that the source code must be available for all identified projects? (Section 2)

We next wish to assess the representative software to determine how well they apply current software development best practices. At this point in the process, to remove potential user/developer bias, we will base our assessment only on publicly available artifacts, where artifacts are the documents, scripts and code that we find in a project's public repository. Example artifacts include requirements, specifications, user manuals, unit test cases, system tests, usability tests, build scripts, API (Application Programming Interface) documentation, READMEs, license documents, process documents, and code. Following best practices does not guarantee popularity, so we will also compare our ranking to how the user community itself ranks the identified projects.

RQ2: Which of the projects identified in RQ1 follow current best practices, based on evidence found by experimenting with the software and searching the artifacts available in each project's repository? (Section 3)

RQ3: How similar is the list of top projects identified in RQ2 to the most popular projects, as viewed by the scientific community? (Section 4)

To understand the state of the practice we wish to learn the frequency with which different artifacts appear, the types of development tools used and the methodologies used for software development. With this data, we can ask questions about how LBM software compares to other research software. That is, in what ways does LBM follow the trends from other developer communities, and in which ways is it different?

RQ4: How do LBM projects compare to research software in general with respect to the artifacts present in their repositories? (Section 5)

RQ5: How do LBM projects compare to research software in general with respect to the use of tools (Section 6) for:

RQ5.a development; and,

RQ5.b project management?

RQ6: How do LBM projects compare to research software in general with respect to principles, processes and methodologies used? (Section 7)

Only so much information can be gleaned by looking at software repositories. To gain additional insight, we need to interview developers. We need to learn their concerns, how they deal with these concerns; we need to learn what pain points exist for them. We wish to know what practices are used by the top LBM projects, so that others can potentially emulate these practices. We also wish to identify new practices that LBM developers can adopt to improve their software in the future. The above points are covered by the questions outlined below:

RQ7: What are the pain points for developers working on LBM software projects? (Section 8)

RQ8: How do the pain points of developers from LBM compare to the pain points for research software in general? (Section 8)

RQ9: For LBM developers what specific best practices are taken to address the pain points and software quality concerns? (Section 9)

RQ10: What research software development practice could potentially address the pain point concerns identified in RQ7). (Section 10)

We investigated the research questions by applying the general methodology summarized in Smith et al. (2021). The specific application of the methodology to LBM is reviewed in Section 2, with the full details in (Michalski, 2021). The current methodology updates the approach used in prior assessments of domains like Geographic Information Systems (Smith et al., 2018b), Mesh Generators (Smith et al., 2016b), Oceanographic Software (Smith et al., 2015), Seismology software (Smith et al., 2018e), statistical software for psychology (Smith et al., 2018d) and medical image analysis software (Dong, 2021). [add citation to medical imaging software, if it is an option —SS]

With our methodology we start by identifying 24 LBM software packages. We then approximately measure the application of best practices for each package by filling in a grading template. Compared with our previous methodology (as used in (Smith et al., 2016b) for instance), the new methodology also includes repository based metrics, such as the number of files, number of lines of code, percentage of issues that are closed, etc. With the quantitative data in the grading template, we rank the software with the Analytic Hierarchy Process (AHP). After this, as another addition to our previous methodology, we interview some of the development teams to further understand the status of their development process. The quantitative and qualitative data is then used to answer the research questions (Sections 2 to 9). Finally, we summarize the threats to validity (Section 11) and the final conclusions and recommendations (Section 12).

2. Methodology

We developed a methodology for evaluating the state of the practice of research software (Smith et al., 2021). This methodology can be instantiated for a specific domain of research software, which in the current case is LBM software. Since we group best practices around conventional software qualities, the first section below provides definitions for the qualities of interest (Section 2.1), along with information on how we assessed these qualities. The following section (Section 2.2) provides an overview of the steps we used to select, measure and compare LBM software. The remaining sections provide details on each step of the steps in the methodology.

2.1. Software Qualities

The following are the software quality definitions used in this exercise, along with comments regarding their measurement.

Installability Installability is measured by the effort required for the installation, uninstallation or reinstallation of a software product in a specified environment (ISO/IEC, 2011; Lenhard et al., 2013). In our case effort includes the time spent finding and understanding the installation instructions, the person-time and resources spent performing the installation procedure, and the absence or ease of overcoming system compatibility issues. An installability score improves if the software has a means to validate the installation and instructions for uninstallation.

Correctness A software program is correct if it behaves according to its stated specifications (Ghezzi et al., 2003, p. 17). This requires that the specification is available. Research software is unlikely to have an explicit specification, since this is not common practice in the field. As a consequence, the correctness of software cannot often be measured directly. We assess correctness indirectly by looking for the following: availability of a requirements specification, reference to domain theory, and explicit use of tools and/or techniques for building confidence, such as documentation generators and software analysis tools.

Verifiability An artifact (like code, or a theory manual) is verifiable if, and only if, every statement therein is verifiable. A statement is verifiable if, and only if, there exists some finite cost-effective process with which a person or machine can check that each statement is correct (adapted from recommended practice for requirements specifications (IEEE, 1998)). Similarly to correctness, verifiability is correlated with the availability of a specification and with reference to domain knowledge. A good measure of verifiability is further correlated with the availability of well written tutorials that include expected output, with software unit testing, and with evidence of continuous integration. In our process we measure correctness and verifiability together.

Reliability Reliability is measured by the probability of failure-free operation of a computer program in a specified environment for a specified time (Ghezzi et al., 2003, p. 357), (Musa et al., 1987). Reliability is thus positively correlated with the absence of errors during installation and use. Recoverability from errors also improves reliability.

Robustness Software possesses the characteristic of robustness if it behaves "reasonably" in two situations: i) when it encounters circumstances not anticipated in the requirements specification; and ii) when the assumptions in its requirements specification are violated (Boehm, 2007), (Ghezzi et al., 2003, p. 19). A good measure of robustness correlates with a reasonable reaction to unexpected input, including data of the wrong type, empty input, or missing files or links. Examples of reasonable reactions include an appropriate error message and the ability to recover the system.

Performance Performance is measured by the degree to which a system or component accomplishes its designated functions within given constraints, such as speed (database response times, for instance), throughput (transactions per second), capacity (concurrent usage loads), and timing (hard real-time demands) (IEEE, 1991; Wiegers, 2003). In this state of the practice assessment performance was not directly quantitatively measured. Instead the documentation of each software package was observed for information alluding to consideration of performance, such testing results, or parallelization instructions.

Usability Usability is measured by the extent to which a software product can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use (Nielsen, 2012). We assumed that high usability correlates with the presence of documentation, including tutorials, manuals, defined user characteristics, and user support. Preferably the user support model will have avenues to contact developers and report issues.

Maintainability A measure of maintainability is the effort with which a software system or component can be modified to correct faults, improve performance or other attributes, and satisfy new requirements (IEEE, 1991; Boehm, 2007). In the current work maintainability is measured by the quality of the documentation artifacts, and the presence of version control and issue tracking. These artifacts can greatly decrease the effort needed to modify software. There are many documentation artifacts that can improve maintainability, including user and developer manuals, specifications, README files, change logs, release notes, publications, forums, and instructional websites.

Modifiability Modifiability refers to the ease with which stable changes can be made to a system and the flexibility of the system to adopt such changes (801, 2017). We did not directly measure modifiability. Instead, developers were asked in interviews if they considered the ease of future changes when developing the software packages, specifically changes to the structure of the system, modules and code blocks. A follow up question is asked if any measures have been taken.

Reusability Reusability refers to the extent to which components of a software package can be used with or without adaptation in other software packages (Kalagiakos, 2003). A good indicator of reusability is a large number of easily reusable components. Therefore, we look for high modularization, which is defined as the presence of smaller components with well defined interfaces. For this state of the practice assessment, we assume that a good measure of reusability correlates with a high number of code files, and the availability of API documentation.

Understandability Understandability is measured by the capability of the software package to enable the user to understand its suitability and function (ISO/IEC, 2001). It is an artifact-dependent quality. Understandability is different for the user-interface, source code, and documentation. In this state of the practice analysis, understandability focuses on the source code. It is measured by the consistency of a formatting style, the extend of modularization, the explicit identification of coding standards, the presence of meaningful identifiers, and the clarity of comments.

Traceability Traceability refers to the ability to link the software implementation and the software artifacts, especially the requirement specification (McCall et al., 1977). Similar to the quality of correctness, this requires the availability of some form of requirements specification. This quality refers to keeping track of information as it changes forms or relates between artifacts. We did not quantitatively measure traceability. Instead, developers were asked in interviews how documentation fits into their development process.

Visibility and Transparency Visibility and transparency refer to the extent to which all of the steps of a software development process, and the current status of it, are conveyed clearly (Ghezzi et al., 2003, p. 32). In this state of the practice assessment a good measure of visibility and transparency correlates with a well defined development process, documentation of the development process and environment, and software version release notes.

Reproducibility Software achieves reproducibility if another developer can take the original code and input data, run it, and obtain the original output (Benureau and Rougier, 2017). We measured reproducibility qualitatively by asking developers if they have any concern that their computational results won't be reproducible, and if they have taken any steps to ensure reproducibility.

Unambiguity Unambiguity refers to the extent to which two readers have similar interpretations when reading software artifacts. In other words, artifacts are unambiguous if, and only if, they only have one interpretation (IEEE, 1998). This state of the practice assessment did not quantitatively measure unambiguity, but we did ask developers if they believe that their current documentation clearly conveys all necessary knowledge to the users. Follow-up questions ask how they achieved unambiguity, or what improvements are needed to achieve it.

2.2. Overall Process

The assessment was conducted via the following steps:

1. List candidate software packages for the domain. (Section 2.3)
2. Filter the software package list. (Section 2.4)
3. Gather the source code and documentation for each software package.
4. Collect quantitative measures from the project repositories. (Section 2.5)
5. Measure using the measurement template. The full measurement template can be found in Smith et al. (2021). (Section 2.5)
6. Use AHP to rank the software packages. (Section 2.6)
7. Interview the developers. (Section 2.7)
8. Domain analysis. (Section 2.8)
9. Analyze the results and answer the research questions.

The above steps depend on interaction with a Domain Expert partner, as discussed in Section 2.9.

2.3. Identify Candidate Software

To answer RQ1 we needed to identify existing LBM projects. The candidate software was found through search engine queries targeting authoritative lists of software. We found LBM software listed on GitHub and swMATH, as well as through scholarly articles. The Domain Expert (Section 2.9) was also engaged in selecting the candidate software.

The following properties were considered when creating the list and reviewing the candidate software:

1. The software functionality must fall within the identified domain.
2. The source code must be viewable.
3. The empirical measures should be available, which implies a preference for GitHub-style repositories.
4. The software cannot be marked as incomplete, or in an initial state of development.

The initial list had 45 packages, including a few that were later found to not have publicly available source code, or to be incomplete projects.

2.4. Filter the Software List

To reduce the length of the list of software packages and to answer RQ1, we filter the initial list of 45. The following filters were applied in the priority order listed.

1. Scope: Software is removed by narrowing what functionality is considered to be within the scope of the domain.
2. Usage: Software packages were eliminated if their installation procedure was missing or not clear and easy to follow.
3. Age: The older software packages (age being measured by the last date when a change was made) were eliminated, except in the cases where an older software package appears to be highly recommended and currently in use.

For the third item in the above filter, software packages were characterized as 'alive' if their related documentation had been updated within the last 18 months. Packages were categorized as 'dead' if the last update was more than 18 month ago.

Filtering by scope, usage, and age decreased the size of the list to 23 packages. Many of the 22 packages that were removed could not be tested as there was no installation guide, they were incomplete, source code was not publicly available, a license was needed, or the project was out of scope or not up to a standard that would support incorporating them into this study. These eliminated software packages are listed in the Appendix of Michalski (2021). Of the remaining 23 packages that were studied, some were kept on the list despite being marked as dead due to their prevalence on authoritative lists on LBM software and due to their surface excellence. After the initial measurement exercise, which took place in mid 2020, we added one more package in January 2022: Musubi. Musubi did not come up in our initial search, but once we learned of it, we felt that this report was incomplete without including it. With the addition of Musubi we did not update the manual collection of data for the previous packages, due to time constraints, but we did redo all data collection that was automated by tools.

The final list of 24 software packages that were analyzed in this project can be found in two tables. Table 1 lists packages that fell into the 'alive' category as of mid 2020 (January 2022 for Musubi), and Table 2 lists packages that were 'dead' at that time. The tables include a hyperlink to each project and, when available, citations for relevant publications. The final list of software packages shows considerable variation in purpose, size, user interfaces, and programming languages. For example, the OpenLB software package is predominantly a C++ package that makes use of hybrid parallelization and was designed to address a range of CFD problems (Heuveline et al., 2009). The software package pyLBM is an all-in-one Python language package for numerical simulations (Graille and Gouarin, 2017). ESPResSo is an extensible simulation package that is specifically for research on soft matter, and is written in C++ and Python (Weik et al., 2019). The HemeLB package is used for efficient simulation of fluid flow in several medical domains, and is written predominantly in C, C++, and Python (Mazzeo and Coveney, 2008).

Name	Released	Updated	Relevant Publication
DL_MESO (LBE)	unclear	2020 Mar	(Seaton et al., 2013)
ESPResSo	2010 Nov	2020 Jun	(Weik et al., 2019)
ESPResSo++	2011 Feb	2020 Apr	(Halverson et al., 2013)
lbmpy	unclear	2020 Jun	(Bauer et al., 2021b)
lettuce	2019 May	2020 Jul	(Bedrunka et al., 2021)
Ludwig	2018 Aug	2020 Jul	(Desplat et al., 2001)
LUMA	2016 Nov	2020 Feb	(Harwood et al., 2018)
MechSys	2008 Jun	2021 Oct	(Galindo-Torres, 2013)
Musubi	2013 Sep	2019 Aug	(Hasert et al., 2014)
OpenLB	2007 Jul	2019 Oct	(Heuveline and Krause, 2010)
Palabos	unclear	2020 Jul	(Latt et al., 2021)
pyLBM	2015 Jun	2020 Jun	
Sailfish	2012 Nov	2019 Jun	(Januszewski and Kostur, 2014)
TCLB	2013 Jun	2020 Apr	(Rokicki and Laniewski-Wollk, 2016)
waLBerla	2008 Aug	2020 Jul	(Bauer et al., 2021a)

Table 1: Alive Software Packages

Name	Released	Updated	Relevant Publication
HemeLB	2007 Jun	2018 Aug	(Mazzeo and Coveney, 2008)
laboetie	2014 Nov	2018 Aug	(Levesque et al., 2013)
LatBo.jl	2014 Aug	2017 Feb	
LB2D-Prime	2005	2012 Apr	
LB3D	unclear	2012 Mar	(Schmieschek et al., 2017)
LB3D-Prime	2005	2011 Oct	
LIMBES	2010 Nov	2014 Dec	
MP-LABS	2008 Jun	2014 Oct	
SunlightLB	2005 Sep	2012 Nov	

Table 2: Dead Software Packages

2.5. Quantitative Measures

We rank the projects by how well they follow best practices (RQ2) via a measurement template, as described in Smith et al. (2021). For each software package (each column in the template), we fill-in the rows of the template. This process takes about 2 hours per package, with a cap of 4 hours. The time constraint is necessary so that the work load is feasible for a team as small as one, given our aim to cap the measurement phase at 160 person hours (Smith et al., 2021). An excerpt of the template, in spreadsheet form, is shown in Figure 2.

Summary Information								
Software name?	DL_MESO	SunlightLB	MP-LABS	LIMBES	LB3D-Prime	LB2D-Prime	laboetie	Musubi
Number of developers	unclear	2	1	unclear	1	1	2	unknown
License?	terms of use	GNU GPL	GNU GPL	GNU GPL	unclear	unclear	GNU GPL	BSD
Platforms?	Windows, OS X, Linux	Linux	Linux	Unix	Windows, Linux	Windows, Linux	Linux	Windows, OS X, Linux
Software Category?	private	public	public	public	public	public	public	public
Development model?	freeware	open source	freeware	freeware	freeware	freeware	unclear	freeware
Programming language(s)?	FORTRAN, C++, Java	C, Perl, Python	FORTRAN, Markdown	FORTRAN	C	C, Shell	FORTRAN, Wolfram Markdown	Fortran
…	…	…	…	…	…	…	…	…
Installability								
Installation instructions?	yes	yes	yes	yes	yes	yes	yes	yes
Instructions in one place?	yes	yes	yes	yes	yes	yes	yes	yes
Linear instructions?	yes	yes	yes	yes	yes	yes	yes	yes
Installation automated?	yes, makefile	yes, makefile	yes, makefile	yes, makefile	yes, makefile	yes, makefile	yes, makefile	yes
Descriptive error messages?	yes	yes	no	n/a	n/a	no	n/a	n/a
Number of steps to install?	8	6	6	4	2	4	4	10
Numbe extra packages?	4	4	3	1	2	2	2	5
Package versions listed?	yes	no	no	no	no	no	no	no
Problems with uninstall?	unavail	unavail	unavail	unavail	unavail	unavail	unavail	unavail
…	…	…	…	…	…	…	…	…
Overall impression (1..10)?	9	7	6	8	7	5	7	8
…	…	…	…	…	…	…	…	…
Surface Reliability								
…	…	…	…	…	…	…	…	…

Figure 2: Excerpt of the Top Sections of the Measurement Template

The full template consists of 108 questions categorized under 9 qualities: (i) installability; (ii) correctness and verifiability; (iii) surface reliability; (iv) surface robustness; (v) surface usability; (vi) maintainability; (vii) reusability; (viii) surface understandability; and, (ix) visibility/transparency.

The questions were designed to be unambiguous, quantifiable and measurable with limited time and domain knowledge. The measures are grouped under headings for each quality, and one for summary information (Figure 2). The summary section provides general information, such as the software name, number of developers, etc. We follow the definitions given by Gewaltig and Cannon (2012) for the software categories. Public means software intended for public use. Private means software aimed only at a specific group, while the concept category is used for software written simply to demonstrate algorithms or concepts. The three categories of development models are: open source, where source code is freely available under an open source license; free-ware, where a binary or executable is provided for free; and, commercial, where the user must pay for the software product.

Several of the qualities use the word “surface”. This is to highlight that, for these qualities in particular, the best that we can do is a shallow measure. For instance, we do not conduct experiments to measure usability. Instead, we are looking for an indication that usability was considered by the developers by looking for cues in the documentation, such as getting started instructions, a user manual or documentation of expected user characteristics.

Tools were used to find some of the measurements, such as the number of files, number of lines of code (LOC), percentage of issues that are closed, etc. The tool GitStats was used to measure each software package’s GitHub

repository for the number of binary files, the number of added and deleted lines, and the number of commits over varying time intervals. The tool Sloc Cloc and Code (scc) was used to measure the number of text based files as well as the number of total, code, comment, and blank lines in each GitHub repository.

As in Smith et al. (2016b), Virtual machines (VMs) were used to provide an optimal testing environment for each package. VMs were used because it is easier to start with a fresh environment without having to worry about existing libraries and conflicts. Moreover, when the tests are complete the VM can be deleted, without any impact on the host operating system. The most significant advantage of using VMs is to level the playing field. Every software install starts from a clean slate, which removes "works-on-my-computer" errors.

2.6. Analytical Hierarchy Process

Once we have measured each package, we still need to rank them to answer RQ2. To do this, we used the Analytical Hierarchy Process (AHP), a decision-making technique that is used to compare multiple options by multiple criteria. AHP was used for comparing and ranking the LBM software packages using the overall impression quality scores that were gathered in the measurement template. AHP performs a pairwise analysis using a matrix and generates an overall score as well as individual quality scores for each software package. (Smith et al., 2016b) shows how AHP is applied to ranking software based on quality measures.

This project used a tool we developed for conducting the AHP analysis. The tool includes a sensitivity analysis to ensure that the software package rankings are appropriate with respect to the uncertainty of the quality scores. For the sensitivity analysis we modified the score by 10% for each package and verified that the overall ranking was stable. The README file of the tool includes requirements and usage information.

2.7. Interview Developers

Several of the research question (RQ5, RQ6, RQ7, RQ9 and RQ10) require going beyond the quantitative data from the measurement template. To gain the required insight, we interviewed developers using a list of 20 questions from Smith et al. (2021). The questions cover the background of the development teams, the interviewees, and the software itself. We ask the developers how they organize their projects. We also ask about their understanding of the users. Some questions focus on the current and past difficulties, and the solutions the team has found, or will try. We also discuss the importance of, and the current situation for, documentation. A few questions are about specific software qualities, such as maintainability, understandability, usability, and reproducibility. The interviews are semi-structured based on the question list; we ask follow-up questions when necessary. Based on our experience, the interviewees often bring up exciting and unexpected ideas.

We requested interviews with developers for all 24 packages, except Munubi, since Munubi was added after our ethics approved had expired. We were able to recruit 4 developers to participate in our study. Our potential interview subjects were found from project websites, code repositories, publications, and biographic information on institution web-pages. Meeting were held using on-line meeting software (either Zoom or Teams), which facilitated recording and automatic transcription. Each meeting lasted about an hour. The interview process was approved by the McMaster University Research Ethics Board under the application number MREB#: 5219.

2.8. Domain Analysis

Since the LBM domain has a reasonably small scope, we are able to view the software as constituting a program family. The concept of a program family is defined by Parnas (1976) as "a set of programs whose common properties are so extensive that it is advantageous to study the common properties of the programs before analyzing individual members". Studying the common properties within a family of related programs is termed a domain analysis.

The domain analysis (also called a commonality analysis) consists of understanding the commonalities, variabilities and common terminology for a program family (Weiss, 1997). Commonalities are goals, theories, models, definitions and assumptions that are common between family members. Variabilities are goals, theories, models, definitions and assumptions that differ between family members. Associated with each variability are its parameters of variation, which summarize the possible values for that variability, along with their potential binding time. The binding time is when the value of the variability is set. It could be set as specification time, build time (when the program is being compiled) or run time (when the code is executing). As an example, for the current assessment, the packages all have the commonality of using LBM techniques to simulate the motion of fluids. For variabilities,

Name	Dim	Pll	Com	Rflx	MFl	Turb	CGE	OS
DL_MESO (LBE)	2, 3	MPI/OMP	Y	Y	Y	Y	Y	W, M, L
ESPResSo	1, 2, 3	CUDA/MPI	Y	Y	Y	Y	Y	M, L
ESPResSo++	1, 2, 3	MPI	Y	Y	Y	Y	Y	L
HemeLB	3	MPI	Y	Y	Y	Y	Y	L
laboetie	2, 3	MPI	Y	Y	Y	Y	Y	L
LatBo.jl	2, 3	-	Y	Y	Y	N	Y	L
LB2D-Prime	2	MPI	Y	Y	Y	Y	Y	W, L
LB3D	3	MPI	N	Y	Y	Y	Y	L
LB3D-Prime	3	MPI	Y	Y	Y	Y	Y	W, L
lbmpy	2, 3	CUDA	Y	Y	Y	Y	Y	L
lettuce	2, 3	CUDA	Y	Y	Y	Y	Y	W, M, L
LIMBES	2	OMP	Y	Y	N	N	Y	L
Ludwig	2, 3	MPI	Y	Y	Y	Y	Y	L
LUMA	2, 3	MPI	Y	Y	Y	Y	Y	W, M, L
MechSys	2, 3	-	Y	Y	Y	Y	Y	L
MP-LABS	2, 3	MPI/OMP	N	Y	Y	N	N	L
Musubi	2, 3	MPI	Y	Y	Y	Y	Y	W, L
OpenLB	1, 2, 3	MPI	Y	Y	Y	Y	Y	W, M, L
Palabos	2, 3	MPI	Y	Y	Y	Y	Y	W, L
pyLBM	1, 2, 3	MPI	Y	Y	N	Y	Y	W, M, L
Sailfish	2, 3	CUDA	Y	Y	Y	Y	Y	M, L
SunlightLB	3	-	Y	Y	N	N	Y	L
TCLB	2, 3	CUDA/MPI	Y	Y	Y	Y	Y	L
waLBerla	2, 3	MPI	Y	Y	Y	Y	Y	L

Table 3: Features of Software Packages (Dim for Dimension (1, 2, 3), Pll for Parallel (CUDA, MPI, OpenMP (OMP)), Com for Compressible (Yes or No), Rflx for Reflexive Boundary Condition (Yes or No), MFl for Multi-fluid (Yes or No), Turb for Turbulent (Yes or No), CGE for Complex Geometries (Yes or No), OS for Operating System (Windows (W), macOS (M), Linux (L)))

the LBM packages can be distinguished by the dimension of problems they simulate, whether or not (and how) they support parallel computation, whether or not compressibility can be considered, whether reflexive boundaries can be employed, whether the simulation handles multi-fluids, whether the simulations handles turbulence, and whether complex geometries can be simulated.

To keep the current presentation focused, we only present the results of our shallow domain analysis. For the LBM domain a table was constructed that distinguishes the programs by their variabilities. In research software the variabilities are often assumptions, as shown in Table 3. The variabilities are listed as columns in Table 3 with the software listed in alphabetical order. A more complete analysis of the commonalities and variabilities between LBM packages is provided in Michalski (2021).

2.9. Interaction With Domain Expert

We partnered with a Domain Expert to vet our list of projects (RQ1) and our ranking (RQ2). The Domain Expert is an important member of the state of the practice assessment team. Pitfalls exist if non-experts attempt to acquire an authoritative list of software, or try to definitively rank the software. Non-experts have the problem that they can only rely on information available on-line, which has the following drawbacks: i) the on-line resources could have false or inaccurate information; and, ii) the on-line resources could leave out relevant information that is so in-grained with experts that nobody thinks to explicitly record it. For the current assessment, our Domain Expert (and paper co-author) is Dr. Zahra Motamed, Assistant Professor of Mechanical Engineering at McMaster University, Hamilton, Ontario, Canada.

The Domain Expert has an important role with verifying the list of LBM packages. In advance of the first meeting with the Domain Expert, they were asked to create a list of top software packages in the domain. This is done to help the expert get in the right mind set in advance of the meeting. Moreover, by doing the exercise in advance, we avoid the potential pitfall of the expert approving the discovered list of software without giving it adequate thought. The Domain Expert was also asked to vet the collected data and analysis. In particular, they were asked to vet the proposed list of software packages and the AHP ranking. These interactions were done via virtual meetings.

3. Ranking Projects Based on Quality Measures

In this section we answer RQ2 by using AHP to rank the identified LBM software against best practices. The sections below summarize how the software packages compare based on the qualities discussed in Section 2.1, as measured via the measurement template (Section 2.5). For the purpose of discussion, the last subsection combines all the qualities into one "global" ranking. The full LBM data is available publicly on Mendeley Data (Smith, 2022b).

3.1. Installability

All of the 24 software packages that were tested have installation instructions. Most have their installation instructions located in one place, often in an instruction manual or on a web-page. Sometimes, like with Ludwig, incomplete installation instructions are found on a home page, with more detailed instructions located on another web-page, or within the documentation. Installability of the software, and maintainability and correctness of the instructions themselves, are improved if all the instructions are in one location.

All 24 packages on the short list can be installed on a Unix-like system. Eight packages could be installed on Windows, and five on macOS. Operating system requirements are listed in the documentation of 20 software packages, but compatible operating system versions are only specified in four packages (ESPResSo, LUMA, Mechsys, Sailfish). All but one of the software packages (TCLB) were tested on Ubuntu for this state of the practice assessment. TCLB was tested on CentOS, since CentOS is mentioned in its installation instructions.

All but one of the software packages (LatBo.jl) have automated at least some of the installation process. Most of these packages, such as waLBerla and SunlightLB, use Make to automate the installation, and a few of them, like lbmpy, use custom scripts.

Errors encountered during the installation process were often quickly fixed thanks to descriptive error messages. Systems that provided vague error messages, such as messages that did not specify which action or file was at fault, were more difficult to troubleshoot. Only three software packages (HemeLB, LB3D, lbmpy) that displayed a descriptive error message were not recoverable, and most of these instances were due to hardware and operating system incompatibility, such as the requirement of CUDA. Fourteen software packages broke during installation. Some packages, such as LB2D-Prime and LB3D-Prime, did not provide a definitive message of the success or failure of installation. In these instances, validating the installation required performing a tutorial or running a script, as described below, if these were available.

About half of the installation instructions are written as if the person doing the installation has none of the dependent packages installed. Unfortunately, software packages, like ESPResSo++, Ludwig, and LUMA, frequently don't list all of their dependencies, or provide only a partial list. Sometimes only an error message during the installation process informs the user of the requirement of these additional packages. A detailed rewrite of the installation instructions from the point of view of installation on a clean operating system is suggested. A clean environment can be achieved for testing purposes by using a virtual machine.

Seventeen software packages require less than 10 dependencies to be installed. All but one software package (LatBo.jl) require less than 20 dependencies. Some packages may automatically install additional dependencies in the background. Twenty of the software packages do not explicitly indicate the versions of software dependencies. Some software package installation issues, specifically those occurring when manual installation of dependencies is required, may be avoided if versions of dependencies are specified. Sixteen software packages do not have detailed instructions for installing dependencies. Sixteen software packages have less than 10 manual installation steps. If dependencies are installed in one command then none of the software packages take more than 20 steps to install. The average number of steps is about eight, and the fewest is two (LB3D-Prime).

All but six (ESPResSo, HemeLB, laboetie, LB3D-Prime, lbmpy, waLBerla) of the software packages have a way to verify the installation. Most have some sort of tutorial examples that can be run by the user. Some other ways of

installation validation include validation scripts (LB2D-Prime, lettuce, Ludwig, LUMA), automatic validation after the installation (LatBo.jl), and instructions to manually review the file system (LIMBES). Uninstallation instructions were found for only one of the software packages: pyLBM.

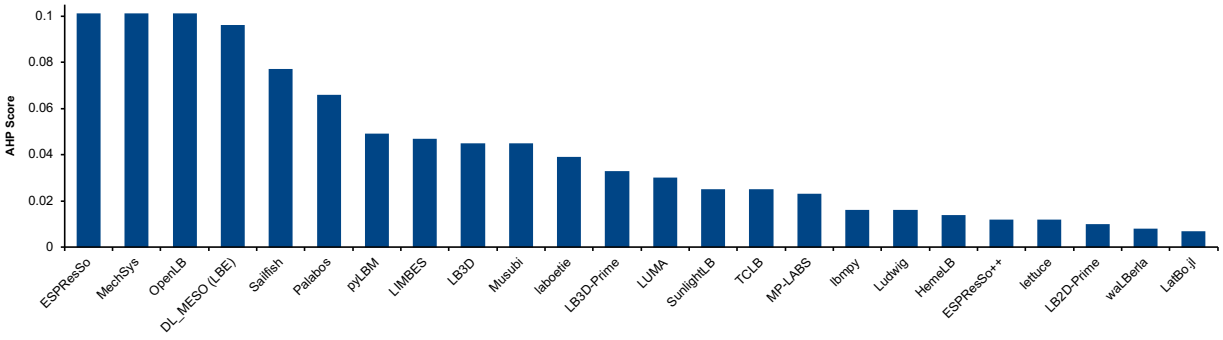

Figure 3: AHP Installability Score

Figure 3 shows the installability ranking of the software packages. Software packages with a higher score (ESPResSo, MechSys, OpenLB, DL_MESO, Palabos) tend to have one set of linear installation instructions written as if the person doing the installation has none of the dependencies installed. The instructions often list compatible operating system versions and include instructions for the installation of dependencies. The top ranked packages often incorporate some sort of automation of the installation process and have fewer manual installation steps. The number of dependencies a package has does not correlate with a higher score. The ability to validate the installation process, often through tutorials or test examples that include expected output, is correlated with a higher score. Active work on a project seems to improve installability, since the top seven ranked packages are all classified as alive.

3.2. Surface Correctness and Verifiability

Seventeen of the software packages explicitly reference domain theory, but a full and rigorous requirements specification, in the software engineering sense (IEEE, 1998; Robertson and Robertson, 1999; ESA, 1991), is absent from all of the projects. Software packages that include a subset of a requirements specification, such as DL_MESO (LBE), keep the information brief and include it within other documents. When present, theory related information is generally found within a user manual, on a web-page, or in a cited publication. In the latter case significant time may be needed to track down the information.

Document generation tools are explicitly used by 13 software packages. Sphinx is used by eight of them, and Doxygen is used by eight. Several of the packages use both.

Tutorials are available for 19 of the software packages. Generally they are linearly written and easy to follow. However, only eight tutorials provide an expected output. It is not possible to verify the correctness of the output of the software packages that are missing this key information. In these cases the user may need to assume correctness if there are no visible errors. A particularly detailed tutorial that include expected output is provided by waLBerla.

Unit tests are only explicitly available for two of the software packages, Ludwig and Musubi. Code modularization of most packages allows for users to create tests with varying degrees of effort. These tests allow developers and users to verify the correctness of fragments of the source code, and in doing so better assess the correctness of the entire package.

The use of continuous integration tools and techniques alludes to a more refined development process where faults are isolated and better recognized. Only three of the packages (ESPResSo, Ludwig, Musubi) mentioned applying the practice of continuous integration in their development process.

Figure 4 shows the ranking for surface correctness and verifiability. Software packages with a higher score tend to have theory documentation. They also explicitly use at least one document generation tool that helps with verification, thus building confidence in correctness. The top ranked software packages all include an easy to follow getting started tutorial, and most of these include expected output. The two top ranked packages, Ludwig and Musubi, indicate the

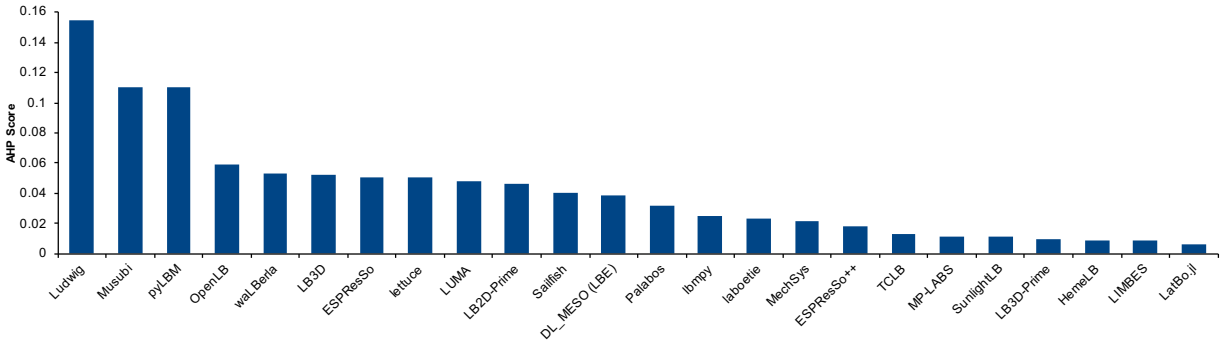

Figure 4: AHP Surface Correctness and Verifiability Score

use of unit testing in their documentation. They also incorporated continuous integration in the development process. Furthermore, eight of the top 10 ranked packages are noted as being alive.

3.3. Surface Reliability

The analysis of surface reliability focused on package installation and tutorials. Errors occurred when installing 16 of the software packages. Every instance prompted an error message. These messages indicated unrecognized commands (even when following the installation guide), missing links, missing dependencies, and syntax errors in code files. In some instances the error messages were vague. Several automatic installation processes could not find or load dependencies. In these instances the installation tried to access outdated external repositories. Seven of the installations were recovered and verified, and one of the installations (LB3D-Prime) was assumed to be recovered due to the absence of any way to verify otherwise. The installation of eight of the software packages could not be recovered. Most of these broken installations could not find external dependencies, encountered system incompatibilities, or displayed vague error messages.

Of the 14 software packages that installed correctly and also have tutorials, four (pyLBM, ESPResSo++, LIMBES, Ludwig) broke during tutorial testing. All of these instances resulted in an error message being displayed. One error (pyLBM) was due to a missing tutorial dependency, another (Ludwig) was due to an invalid command despite following the tutorial, and the final two errors were vague execution errors. Of the four broken tutorial instances, only the missing dependency one was recoverable.

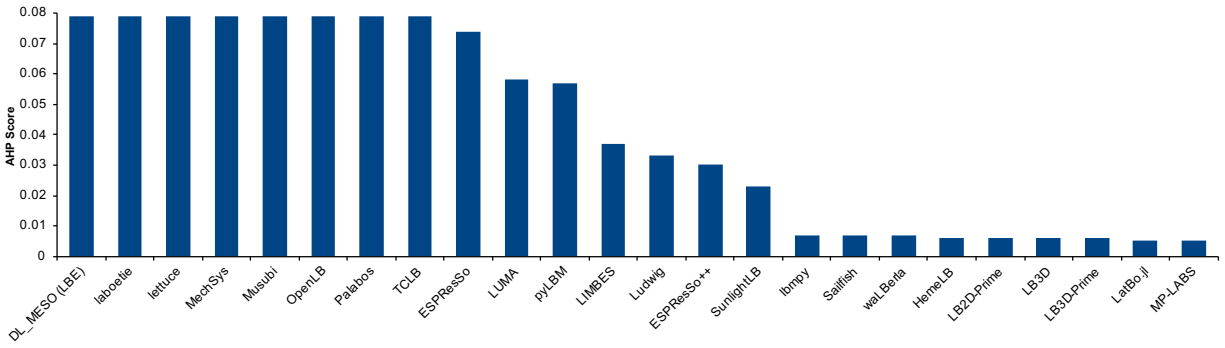

Figure 5: AHP Surface Reliability Score

Figure 5 shows the surface reliability ranking of the software packages. Software packages with a high score either did not break during installation, or the broken installation was recoverable. All of the top five ranked packages

have tutorials. The package pyLBM broke during tutorial testing, but a descriptive error message helped in recovery. Nine of the top 10 ranked packages are noted as being alive.

3.4. Surface Robustness

The software packages were tested for handling unexpected input, including incorrect data types, empty input, and missing files or links. Success is predicated on a reasonable response from the system, including appropriate error messages and an absence of unrecoverable system failures.

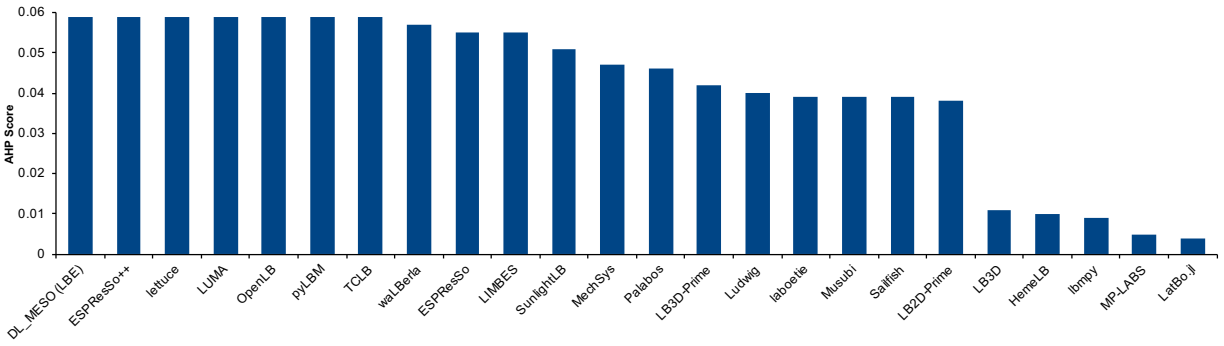

Figure 6: AHP Surface Robustness Score

Figure 6 shows the surface robustness ranking of the software packages. Software packages with a high score behaved reasonably in response to unexpected input as described above. All of the software packages that installed correctly passed this test. They output descriptive error messages or at least did not crash. Software packages with a lower surface robustness score did not install correctly, so their robustness score may not be a true reflection of runtime robustness. All successfully installed software packages that require a plain text input file correctly handled an unexpected change to these input files, including a replacement of new lines with carriage returns. Furthermore, nine of the top 10 ranked packages are noted as being alive.

3.5. Surface Performance

Although the software packages all apply LBM to solve scientific computing problems, the packages focus on varied CFD problems, with varying parameters, and are technically different from each other (as shown in Table 3). Due to this, a comparison of run-time performance is not appropriate. Instead we looked through each software package's artifacts for evidence that performance was considered. The artifacts of 18 software packages mentioned parallelization. This included GPU processing and the CUDA parallel computing platform, which were mentioned in the artifacts of 6 packages (ESPResSo, lbmpy, lettuce, pyLBM, Sailfish, TCLB). When a high degree of parallelization is possible GPUs provide superior speed compared to CPUs. The software package TCLB is implemented in a highly efficient multi-GPU code to achieve performance suitable for model optimization (Rutkowski et al., 2020). In the Ludwig package, a so-called mixed mode approach is used where fine-grained parallelism is implemented on the GPU, and MPI is used for even larger scale parallelism (Gray and Stratford, 2013). While one software package (Sailfish) required CUDA and GPU processing, some (ESPResSo, lbmpy, lettuce, pyLBM, TCLB) have the option of using either the GPU or the CPU. In general, the packages that require GPU and CUDA have better performance at the expense of installability and surface reliability.

3.6. Surface Usability

Software package artifacts were reviewed for the presence of a tutorial, a user manual, documented user characteristics, and a user support model. In total 19 software packages have a tutorial, 13 have a user manual, and 11 have both. The tutorials vary in scope and substance, and eight include expected output. Most user manuals are in the form of a file that can be downloaded, while some are rendered on a web-page. Some packages (waLBerla) do

not have a user manual, but do have useful documentation distributed throughout their web-pages. Expected user characteristics are documented in five software packages (laboetie, LIMBES, Ludwig, Musubi, Palabos). Users are typically scientists or engineers. Their background is often physics, chemistry, biophysics, or mathematics. All but one of the packages (LIMBES) have a user support model, and many of them have multiple avenues of user support. The most popular avenue of support is Git, followed by email and forums. One software package (OpenLB) has an FAQ page.

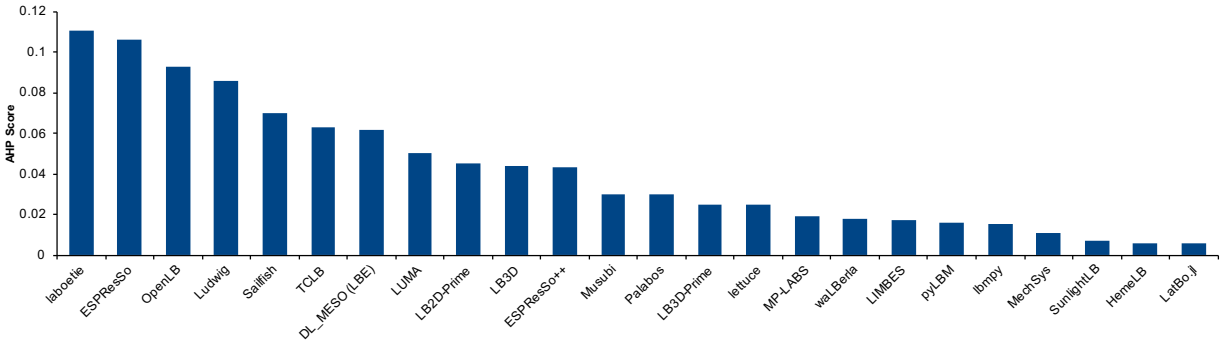

Figure 7: AHP Surface Usability Score

Figure 7 shows the surface usability ranking of the software packages. Software packages with a high score have a tutorial and user manual, sometimes have documented user characteristics, and have at least one user support model. Many packages have several user support models. Furthermore, four of the top five packages are noted as being alive.

3.7. Maintainability

Software packages were reviewed for the presence of artifacts. Every type of artifact or file that is not a code file was recorded. (The list of artifacts will be discussed further in Section 5.) The software packages were also reviewed for software release and documentation version numbers. This information can be used to troubleshoot issues and organize documentation. All but three software packages (LatBo.jl, LB3D-Prime, MechSys) have source code release and documentation version numbers.

When present, information on how code is reviewed, or how to contribute to the project was also noted. In total, 11 software packages have this information, which was found in various artifacts, including in developer guides, contributor guides, user guides, developer web-pages, and README files.

Issue tracking is used in 23 software packages, 15 of which use GitHub or GitLab to host their git repository, seven use email, and one (SunlightLB) uses SourceForge. Most software packages that use GitHub/GitLab have the majority of their issues closed, and only three (laboetie, lettuce, Sailfish) have less than 50 percent of their issues closed. All of the top five overall ranked packages have most of their issues closed. Alive packages (11 use issue tracking) have 64% of their issues closed, while dead packages (3 use issue tracking) have 71% of their issues closed. This information is presented in Table 4. With respect to the host for the git repository, 13 packages use GitHub and two (Palabos, waLBerla) use GitLab. Of the other packages, one package (SunlightLB) uses CVS for issue tracking and version control, and seven packages do not appear to use any issue tracking system.

Software package code files were further measured for the percentage of code that is comments. The findings are presented in Table 4, sorted by the percentage. Packages with a higher percentage of comments were assumed to be more maintainable. Comments represent more than 10 percent of code files in 16 packages, and the average percentage of code comments is about 14 percent. All of the top five overall ranked packages have more than the average. LUMA has only 0.2 percent comments, the fewest of any package. This package has the most lines of source code, with over four million. The next largest package is ESPResSo++ with one million.

Figure 8 shows the maintainability ranking of the software packages. Software packages with a high score provide version numbers on documents and source code releases, have an abundance of high quality artifacts, and use an issue tracking tool and version control system. These packages also appear to manage their issue tracking, having most of

Name	% Issues Closed	% Code Comments	Status
MP-LABS	100.00	26.67	Dead
Musubi	Not Git	24.19	Alive
waLBerla	72.90	22.62	Alive
OpenLB	Not Git	22.43	Alive
ESPResSo	89.26	21.78	Alive
Ludwig	60.00	20.70	Alive
Palabos	89.47	17.76	Alive
SunlightLB	Not Git	17.67	Dead
LIMBES	Not Git	17.39	Dead
ESPResSo++	66.28	17.10	Alive
HemeLB	No Issues	16.68	Dead
pyLBM	66.67	16.12	Alive
MechSys	Not Git	15.11	Alive
LB3D-Prime	Not Git	14.34	Dead
LB3D	Not Git	13.76	Dead
LB2D-Prime	Not Git	13.61	Dead
Sailfish	22.22	9.26	Alive
lettuce	33.33	8.19	Alive
DL_MESO (LBE)	Not Git	8.06	Alive
TCLB	60.32	6.02	Alive
laboetie	18.75	2.47	Dead
lbmpy	58.33	2.03	Alive
LatBo.jl	93.33	0.40	Dead
LUMA	85.71	0.20	Alive

Table 4: Git Repository Data, Sorted by Percentage of Code that is Comments

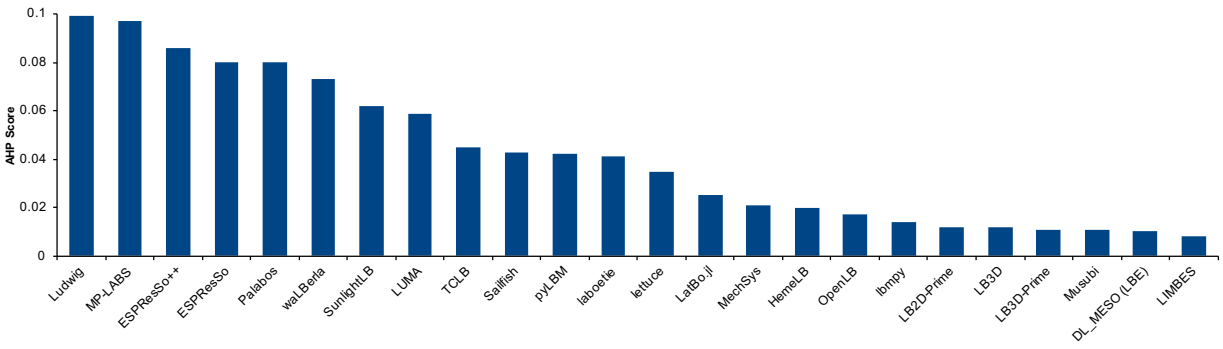

Figure 8: AHP Maintainability Score

their issues closed. Their code files are well commented with more than 10 percent of the code being comments. Four of the top 5 ranked packages are noted as being alive.

3.8. Reusability

We measured the total number of source code files for each project. We assume that a large number of source files is associated with increased reusability, due to our assumption that this indicates increased modularization. Some packages have more features than others. This is assumed to contribute to reusability, since they have more source code for potential reuse. The software packages were also reviewed for the presence of API documentation, which indicates that a software package was developed with interaction between other software applications in mind.

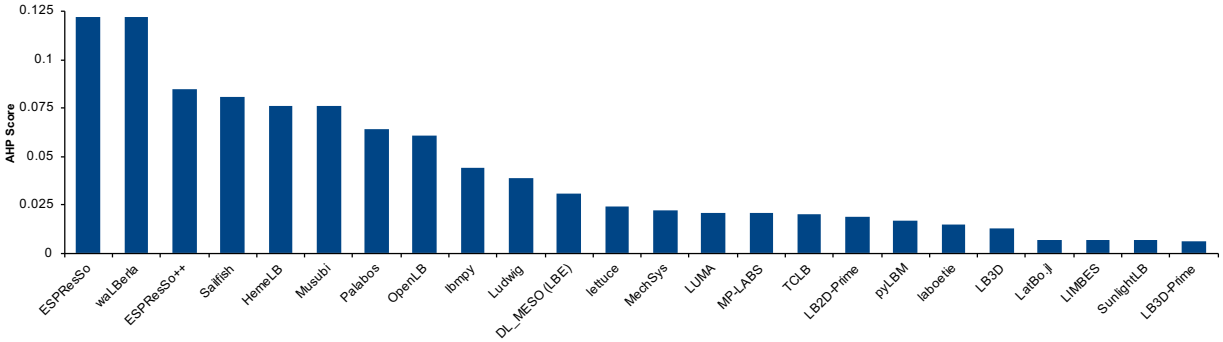

Figure 9: AHP Reusability Score

Figure 9 shows the reusability ranking of the software packages. Software packages with a high score have thousands of source code files and API documentation. The highest scoring packages, ESPResSo and waLBerla, have extensive functionality, including graphical visualizations and non-LBM modelling. For this reason, a comparison with other software packages is not on a level field. However, these packages do have an abundance of reusable components. Nine of the top 10 ranked packages are noted as being alive.

Table 5 shows file and Lines Of Code (LOC) data for the software packages. Packages with a high reusability score do not have as many LOC per text file, generally having a few hundred lines or less. This suggests that the source code of these packages is likely functionally modularized, and modules could be reused in other projects.

The package waLBerla scored high on reusability because of its focus on modularity. The modularity in the waLBerla framework was included to enhance productivity, reusability, and maintainability (Bauer et al., 2021a). The design of waLBerla has enabled it to be successfully applied in several projects as a basis for various extensions (Bauer et al., 2021a).

Name	Text Files	Binary Files	LOC	Avg. LOC / Text File
LUMA	314	19	4399723	14011
LatBo.jl	41	0	42172	1029
LB2D-Prime	82	19	54755	668
LB3D-Prime	23	6	12944	563
DL_MESO (LBE)	310	51	170223	549
LB3D	99	76	39766	402
laboetie	133	1	48403	364
waLBerla	2643	69	873988	331
MechSys	333	3	95707	287
Palabos	1974	71	563841	286
lbmpy	250	29	61632	247
SunlightLB	36	1	7646	212
Musubi	1347	1839	281879	209
LIMBES	26	1	4872	187
Ludwig	954	32	162518	170
OpenLB	1438	7	218406	152
ESPResSo	1390	83	195083	140
MP-LABS	307	3	43124	140
pyLBM	272	108	37234	137
ESPResSo++	1406	30	165194	118
HemeLB	1102	44	123806	112
Sailfish	632	11	69398	110
lettuce	73	1	7660	105
TCLB	594	7	49156	83

Table 5: Module Data

3.9. Surface Understandability

Ten random source code files of each software package were reviewed for several measures. This assessment of surface understandability may not perfectly reflect each package, due to the practical limitation of only examining 10 files.

All of the packages appear to have consistent indentation and formatting. Only HemeLB, LUMA, and Musubi explicitly identify coding standards that are used during development. Generally, the software packages use consistent, distinctive, and meaningful code identifiers. Only four packages (LB2D-Prime, LB3D-Prime, LIMBES, MP-LABS) appear to use vague identifiers, such as single letters for variables. Symbolic constants were observed in the source code of 13 packages. The constants are used for various parameters, mathematical constants, and matrix definitions. All of the packages are reasonably well commented, with the comments clearly indicating what is being done (as opposed to how it is being done). Domain algorithms are noted in the source code of 11 packages. Table 5 suggests that the software packages are modularized to various degrees. When observing the source code files, it was found that 14 of the packages have a consistent style and order of function parameters.

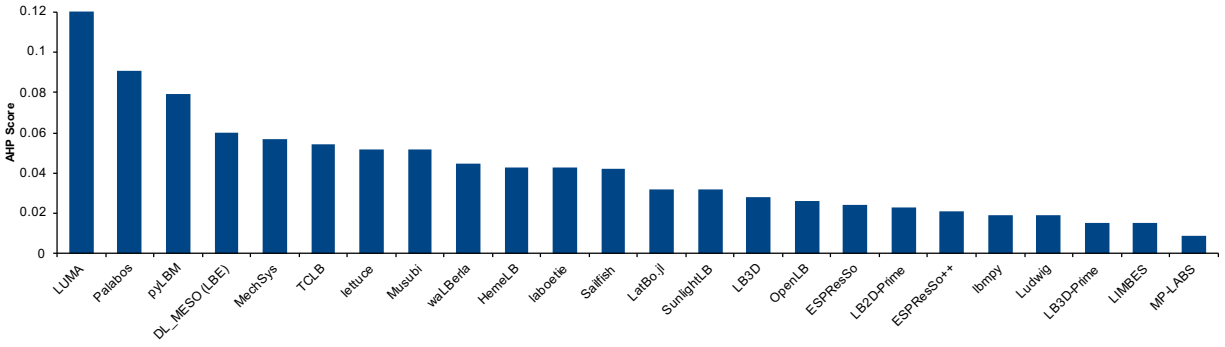

Figure 10: AHP Understandability Score

Figure 10 shows the surface understandability ranking. Software packages with a high score have a consistent indentation and formatting style, and consistent, distinctive, and meaningful code identifiers. They also have symbolic constants, and explicitly identify mathematical and LBM algorithms. Their comments are clear and indicate what is being done in the source code. The source code is well modularized and structured. All of the top five ranked packages are noted as being alive.

3.10. Visibility and Transparency

Software package artifacts were reviewed for the identification of a specific development model, like a waterfall of agile development model, and the presence of documentation recording the development process and standards. They were also reviewed for the identification of the development environment, and the presence of release notes. The packages tended to not explicitly use well-known development models. This was also noted in the interviews with developers (Section 7). The development teams of these packages are fairly small and easily organized without the need for such processes. Seven of the software packages did have some artifacts outlining the general development process, how to contribute, and the status of the package or its components. Eight of the packages have artifacts that note the development environment. While this information could help developers, and would improve transparency, the small close-knit nature of the development teams make explicitly publicly specifying this information practically unnecessary. Version release notes were found in nine of the software packages.

Figure 11 shows the visibility and transparency ranking of the software packages. Software packages with a high score have an explicit development model and defined development process. They also have detailed and easy to access notes accompanying software releases. Four of the top five ranked packages are noted as being alive.

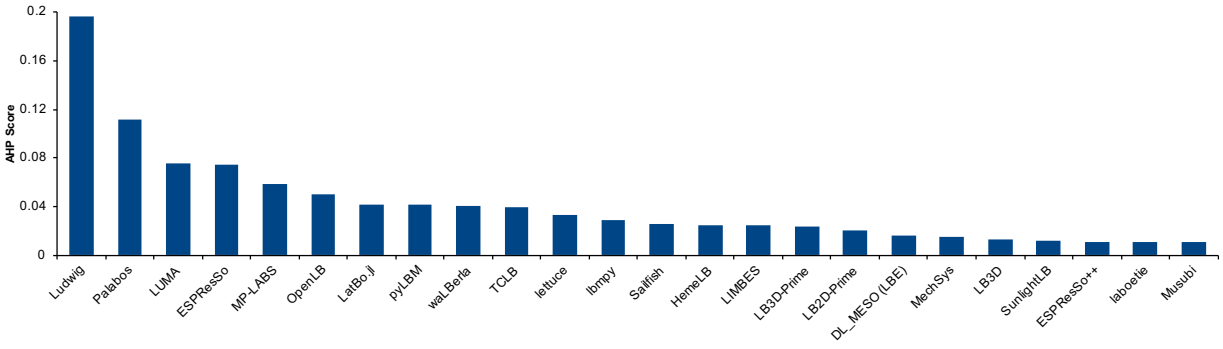

Figure 11: AHP Visibility and Transparency Score

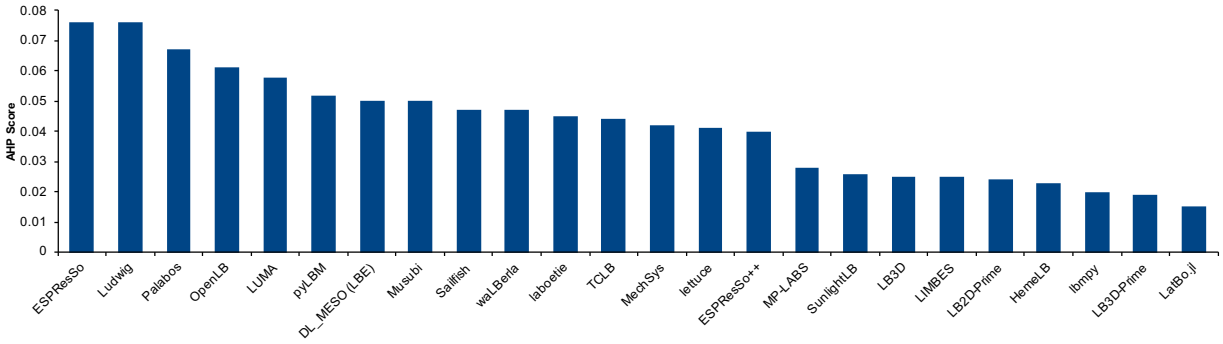

Figure 12: AHP Overall Score

3.11. Overall Quality

Figure 12 shows the overall ranking of the software packages. In the absence of other information on priorities, the overall ranking was calculated by assuming an equal weighting between all qualities. If the qualities were weighed differently, the overall software package ranking would change. Software packages with an overall high score ranked high in at least several of the individual qualities that were quantitatively measured.

Looking at the top three ranked packages: ESPResSo achieved a relatively high score in installability, surface usability, maintainability, reusability, and visibility and transparency. Ludwig scored high in surface correctness and verifiability, surface robustness, surface usability, maintainability, and visibility and transparency. Palabos scored high in installability, surface reliability, surface robustness, maintainability, understandability, and visibility and transparency.

4. Comparison to Community Ranking

To address RQ3, Table 6 compares our LBM software package rankings against their popularity in the research community, as estimated by repository stars and watches. Nine packages do not use GitHub, so they do not have a measure of repository stars. Looking at the repository stars of the other 15 packages, we can observe a pattern where packages that have been highly ranked by our assessment tend to have more stars than lower ranked packages. Our best ranked package (ESPResSo) has the second most stars, while our ninth ranked package (Sailfish) has the highest number of stars. The same correlation is observed in the repository watch column, although this column contains less data, since two of the packages (Palabos, waLBerla) use GitLab, which does not track watches. Consistent with the star measure, our ninth ranked package (Sailfish) has the highest number of watches and our best ranked package (ESPResSo) has the second most watches. Packages designated as lower quality often do not use GitHub or GitLab,

Name	Our Ranking	Repository Stars	Repository Star Rank	Repository Watches	Repository Watch Rank
ESPResSo	1	145	2	19	2
Ludwig	2	27	8	6	7
Palabos	3	34	6	GitLab	GitLab
OpenLB	4	No Git	No Git	No Git	No Git
LUMA	5	33	7	12	4
pyLBM	6	95	3	10	5
DL_MESO (LBE)	7	No Git	No Git	No Git	No Git
Musubi	8	No Git	No Git	No Git	No Git
Sailfish	9	186	1	41	1
waLBerla	10	20	9	GitLab	GitLab
laboetie	11	4	13	5	8
TCLB	12	95	3	16	3
MechSys	13	No Git	No Git	No Git	No Git
lettuce	14	48	4	5	8
ESPResSo++	15	35	5	12	4
MP-LABS	16	12	11	2	9
SunlightLB	17	No Git	No Git	No Git	No Git
LB3D	18	No Git	No Git	No Git	No Git
LIMBES	19	No Git	No Git	No Git	No Git
LB2D-Prime	20	No Git	No Git	No Git	No Git
HemeLB	21	12	11	12	4
lbmpy	22	11	12	2	9
LB3D-Prime	23	No Git	No Git	No Git	No Git
LatBo.jl	24	17	10	8	6

Table 6: Repository Ranking Metrics

or have only a few stars/watches. Although the details of the rankings differ, our measures show that following best practices tends to correlate with increased popularity in the scientific community.

Notable exceptions to this trend occur for three projects that rank high by star count, but relatively low in the "follow best practices" ranking: TCLB (star rank 3, our rank 12), lettuce (star rank 4, our rank 14) and ESPResSo++ (star rank 5, our rank 15). Part of the discrepancy is because all projects are ranked by our process, but not all projects are measured by stars. Also, our process equally weights all qualities (Section 3.11), but for real users some qualities will be more important than others. Discrepancies between our measures and the star measures may also be due to inaccuracy with using stars to approximate popularity, because of how people use stars, because young projects have less time to accumulate stars, and because stars represent the community's feeling in the past more than current preferences (Szulik, 2017). Older packages (like ESPResSo++ and TCLB) have had more time to accumulate stars and watches. Young projects have less time to accumulate stars, like HemeLB, which only has 12 stars (at the time of measurement) having recently moved from a different GitHub repository. Lettuce is a young project (released in 2019) with a relatively high number of stars. The discrepancy in rankings here may be due to a mismatch between the time when lettuce was manually measured by our template and when the stars were counted. The late addition of Musubi meant a large gap between when projects were manually measured and the time when our automated data (like stars) was collected. A final reason for inconsistencies between our ranking and the community's ranking is that, as for consumer products, more factors influence popularity than just quality.

Although our ranking and the estimate of the community's ranking are not perfect measures, they do suggest a correlation between best practices and popularity. We do not know which comes first, the use of best practices or popularity, but we do know that the top ranked packages tend to incorporate best practices. The next sections will explore how the practices of the LBM community compare to the broader research software community. We will also

Common	Less Common	Rare
Authors / Developers List	Change Log / Release Notes	API Documentation
Issue Tracker	Design Documentation	Developer / Contributor Manual
Library Dependency List	Functional Spec. / Notes	FAQ / Forum
Installation Guide / Instructions	Performance Information	Verification and Validation Plan
Theory Notes	Test Plan / Script / Cases	Video Guide (including YouTube)
List of Related Publications	User Manual / Guide	Requirements Spec.
Makefile / Build File		
README File		
License		
Tutorial		
Version Control		

Table 7: Artifacts Present in LBM Packages, Classified by Frequency

investigate the practices from the top projects that others within the LBM community, and within the broader research software community, can potentially adopt.

5. Comparison Between LBM and Research Software for Artifacts

As part of filling in the measurement template the LBM packages were examined for the presence of artifacts, which were then categorized by frequency. We have grouped them into common, less common, and rare artifacts in Table 7. Common artifacts were found in 16 to 24 (>63%) of the software packages. Less common artifacts were found in 8 to 15 (30-63%) of the software packages. Rare artifacts were found in 1 to 7 (<30%) of the software packages. In this section we answers RQ4 by comparing the artifacts that we observed in LBM repositories to those observed and recommended for research software in general. Our comparison may point out areas where some LBM software packages fall short of current best practices. This is not intended to be a criticism of any existing packages, especially since in practice not every project needs to achieve the highest possible quality. However, rather than delve into the nuances of which software can justify compromising which practices we will write our comparison under the ideal assumption that every project has sufficient resources to match best practices.

The top four packages, ESPREsSo, Ludwig, Palabos, and OpenLB, have all of the common artifacts, except Palabos does not have theory notes and only three of them (ESPREsSo, Ludwig, Palabos) appear to use a version control system. The top four packages also have most of the less common artifacts. At the time of data collection, only one of the four packages (Palabos) did not have a user manual or guide, but there was a broken link on the package website indicating that such an artifact might exist. This broken link was later fixed, but this is not reflected in our data because it was not present at the time of data collection (mid 2020). Despite the broken link, Palabos does have a detailed and informative website. As far as we could observe, Ludwig does not appear to have a publicly visible design and OpenLB does not appear to use a version control system. However, OpenLB's use of version numbers suggests that version control may be used privately by the developers.

The top four ranked packages do not have many of the rare artifacts. None of the top four packages have any explicit API documentation. Three of these packages (ESPREsSo, Ludwig, Palabos) have information on contributing to the project. Two of them (OpenLB, Palabos) have an FAQ section or forum. One (OpenLB) has verification and validation notes, and a video guide for the software.

The majority of LBM generated artifacts correspond to general recommendations from research software developers. A union of the three columns in Table 7 mostly corresponds to recommendations made by the research software community. Most of the items in Table 7 are included in guidelines for writing research software, such as the DLR (German Aerospace Centre) Software Engineering Guidelines (Schlauch et al., 2018), xSDK (Extreme-scale Scientific Software Development Kit) Community Package Policies (Smith et al., 2018a), the EURISE (European Research Infrastructure Software Engineers') Network Technical Reference Thiel (2020) and CLARIAH (Common Lab Research Infrastructure for the Arts and Humanities) Guidelines for Software Quality (van Gompel et al., 2016). As an

example, LBM developers commonly list the external libraries on which the software depends, as recommended by Brett et al. (2021), Smith et al. (2018a) and van Gompel et al. (2016). Although in the rare column, some LBM developers include an FAQ (Frequently Asked Questions) document as recommended by (Orviz et al., 2017), (Thiel, 2020) and (van Gompel et al., 2016). LBM developers also follow the ubiquitous recommendation to have a README file, although, as for other software, the content of README files is not standardized (Prana et al., 2018).

Three of the items that appear in Table 7 do not explicitly appear in the software development guidelines: i) list of related publications, ii) performance information and iii) video guides. The occurrence of the first two items in this list are likely more a consequence of how we collected our list of LBM artifacts, rather than being a unique practice of LBM developers. We highlighted publications and performance information because we were explicitly looking for both as part of filling in our measurement template. Related publications and performance information certainly appear in other research software projects; the practices just aren't highlighted in guidelines because they are implicit in other documentation recommendations, such as theory documentation and user guides. The observation of a video guide artifact is more novel. Fogel (2005) recommends videos for open source projects, and other projects likely use them, but the practice has apparently not yet made it into research software community's guidelines. Putting aside the debate of the effectiveness of learning from print versus video, video documentation is growing in popularity, with a majority of Generation Z learners preferring video to print (Genota, 2018).

Although the LBM community participates in most of the practices we found listed in the general research software guidelines, some recommended practices were not observed or rarely observed. For instance API documentation was rarely observed for LBM software, but it is a frequently recommended artifact (Smith et al., 2018a; Thiel, 2020; van Gompel et al., 2016; Orviz et al., 2017; Institute, 2022; Zadka, 2018). In addition to the items in the last column of Table 7, we can add the following community recommended items that we rarely, if ever, observed:

- A roadmap showing what is planned for the future (Yehudi, 2021; Thiel, 2020; van Gompel et al., 2016). We did see this information mentioned in some repos, but we did not observe a specific artifact devoted to this purpose.
- A code of conduct to explicitly say how developers should treat one another (Yehudi, 2021; Thiel, 2020). We did not observe a policy of this sort, possibly because the number of developers for each LBM project is small.
- Programming style guidelines so that programming labels and formatting are consistent (Schlauch et al., 2018; Thiel, 2020; Orviz et al., 2017; Zadka, 2018; van Gompel et al., 2016; Wilson et al., 2014). We saw this information as part of some developer guides, but only rarely.
- Checklists can be used in projects to ensure that best practices are followed by all developers. Some examples include checklists merging branches into master (Brown, 2015), checklists for saving and sharing changes to the project (Wilson et al., 2016), checklists for new and departing team members (Heroux and Bernholdt, 2018), checklists for processes related to commits and releases (Heroux et al., 2008) and checklists for overall software quality (Thiel, 2020; Institute, 2022). For LBM software, ESPResSo has a checklist for managing releases, but otherwise they are used only rarely for LBM.
- Uninstall instructions (van Gompel et al., 2016) were not observed for any of the LBM projects.

The above discussion shows that, taken together, LBM projects fall short of recommended best practices for research software. However, LBM software is not alone in this. Many, if not most, research projects fall short of best practices. Software requirements documentation provides a good example of the common deviation between recommendations and practice. Although requirements documentation is recommended by some (Schlauch et al., 2018; Heroux et al., 2008; Smith and Koothoor, 2016), in practice research software developers often do not produce a proper requirements specification (Heaton and Carver, 2015). Sanders and Kelly (2008) interviewed 16 scientists from 10 disciplines and found that none of the scientists created requirements specifications, unless regulations in their field mandated such a document. Nguyen-Hoan et al. (2010) showed requirements are the least commonly produced type of documentation for research software in general. When looking at the pain points for research software developers, Wiese et al. (2019) found that software requirements and management is the software engineering discipline that most hurts scientific developers, accounting for 23 % of the technical problems reported by study participants. The lack of support for requirements is likely due to the perception that up-front requirements are impossible for research software

Development Tools	Dependencies	Project Management Tools
Continuous Integration	Build Automation Tools	Collaboration Tools
Code Editors	Technical Libraries	Email
Development Environments	Domain Specific Libraries	Change Tracking Tools
Runtime Environments		Version Control Tools
Compilers		Document Generation Tools
Unit Testing Tools		
Correctness Verification Tools		

Table 8: Observed development tools, dependencies and project management tools

(Carver et al., 2007; Segal and Morris, 2008), but when the instance on "up-front" requirements is dropped, allowing the requirements to be written iteratively and incrementally, requirements are feasible (Smith, 2016).

6. Comparison of Tool Usage Between LBM and Other Research Software

Software tools are used to support the development, verification, maintenance, and evolution of software, software processes, and artifacts (Ghezzi et al., 2003, p. 501). Many tools are used by LBM software packages, as summarized in Table 8. The tools are subdivided into development tools, dependencies, and project management tools. In this section we answer RQ5 by comparing aspects of the tool usage in LBM to their utilization in the research software community in general.

Development tools support the development of end products, but do not become part of them, unlike dependencies that remain in the application once it is released (Ghezzi et al., 2003, p. 506). Although not shown in Table 8, debuggers were also likely used. Only three (ESPResSo, Ludwig and Musubi) packages mentioned continuous integration tools, like Travis CI. Code editors and compilers were explicitly noted to have been used by several packages, and were likely used by all of them. One of the packages (Ludwig) explicitly noted the use of proprietary unit testing code written in C. Likewise, the use of proprietary code for verifying the correctness of output was noted by one of the developers (pyLBM). Similar tools were likely used when developing the other software packages.

For the dependency tools (Table 8), we observed that most of the software packages use some sort of build automation tools, most commonly Make. They all use various technical and domain specific libraries. Technical libraries include visualization (e.g. Matplotlib, ParaView, Pygame, VTK), data analysis (e.g. Anaconda, Torch), and message passing libraries (e.g. MPICH, Open MPI, PyZMQ). Domain specific libraries include research software libraries (e.g. SciPy).

Many of the software packages that were assessed were developed by teams of two or more people. Their work needed to be coordinated and managed. Table 8 shows the types of project management tools that were explicitly noted in the artifacts, web-pages, or interviews with the developers. As with development tools and dependencies, it is possible that other types of project management tools are used, but they were not visible in the artifacts to which we had access. Collaboration tools, most often email and video, are noted as being used when developing software projects. Project management software was not explicitly mentioned during our interviews, but it is possible that some of the projects use such software. Many of the projects are located on GitHub, where the developers use the platform to help manage their projects, especially bug related issues. Most of the projects appear to use change tracking and version control tools. Document generation tools are mentioned in 12 of the 24 projects. The tools Sphinx and Doxygen are explicitly used in this capacity.

The poor adoption of version control tools that Wilson lamented in 2006 (Wilson, 2006) has greatly improved in the intervening years. From Section 3.7, 67% of LBM packages use version control (GitHub, GitLab or CVS). (From Table 4, 11/15 or 73% of alive packages use version control.) The proliferation of version control tools for LBM matches the increase in the broader research software community. A little over 10 years ago version control was estimated to be used in only 50% of research software projects (Nguyen-Hoan et al., 2010), but even at that time Nguyen-Hoan et al. (2010) noted an increase from previous usage levels. A survey in 2018 shows 81% of

developers use a version control system (AlNoamany and Borghi, 2018). Smith (2018) has similar results, showing version control usage for alive projects in mesh generation, geographic information systems and statistical software for psychiatry increasing from 75%, 89% and 17% (respectively) to 100%, 95% and 100% (respectively) over a four year period ending in 2018. (For completeness the same study showed a decrease in version control usage for seismology software over the same time period, from 41% down to 36%). Almost every software guide cited in Section 5 includes the advice to use version control. The high usage of version control tools in LBM software matches the trend in research software in general.

As mentioned in Section 3.2, continuous integration is rarely used in LBM (3 of 24 packages or 12.5%). This contrasts with the frequency with which continuous integration is recommended in research software development guidelines (Brett et al., 2021; Brown, 2015; Thiel, 2020; Zadka, 2018; van Gompel et al., 2016). We could not find published data on the frequency with which continuous integration is used in general research software. Our impression is that like LBM software, other research software packages lag behind the recommended best practice of employing continuous integration.

Interviews with the developers revealed a potentially more frequent use of both unit testing and continuous integration, compared to what was observed from studying the repository artifacts.

7. Comparison of Principles, Process and Methodologies to Research Software in General

This section answers research question RQ6 by comparing the principles, processes and methodologies used for LBM software to what can be gleaned from the literature on research software in general. In our data collection for LBM software, the software development process is not explicitly indicated in the artifacts for most of the packages. However, during our interviews one developer (ESPResSo) told us their non-rigorous development model is like a combination of agile and waterfall. Employing a loosely defined process makes sense for LBM software, given that the teams are generally small and self-contained. Although eleven of the packages explicitly convey that they would accept outside contributors, generally the teams are centralized, often working at the same institution. Working at the same institution means that an informal process can show success, since informal conversations are relatively easy to have.

Interviews with developers confirmed a similar project management processes. In teams of only a couple of developers, additions of new features or major changes are discussed with the entire team. Projects with more than a couple developers have lead developer roles. These lead developers review potential additions to the software. One of the developers (ESPResSo) that was interviewed noted that an ad hoc peer review process is used to assess major changes and additions. Using peer review (also called technical review) matches with recommended practice for research software (Heroux et al., 2008; Givler, 2020; Orviz et al., 2017; USGS, 2019).

Two types of software changes were discussed during interviews with developers. One is feature additions, which arise from a scientific or functional need. These changes involve formal discussions within the development team, and lead developer participation is mandatory. The other change type is code refactoring, which only sometimes involves formal discussions with the development team. New developers were noted to play an increased role in these changes compared to the former changes. Software bugs are typically addressed in a similar fashion as code refactoring. Issue tracking is commonly used to manage these changes.

Our observations of an informally defined process, with elements of agile methods, matches what has been observed for research software in general. Scientific developers naturally use an agile philosophy (Ackroyd et al., 2008; Carver et al., 2007; Easterbrook and Johns, 2009; Segal, 2005; Heaton and Carver, 2015), or an amethododical process (Kelly, 2013), or a knowledge acquisition driven process (Kelly, 2015).

Most of the software packages do not explicitly state the motivations or design principles that were considered when developing the software. One package, Sailfish, indicates in its artifacts that shortening the development time was considered in early stages of design, with the developers using Python and CUDA/OpenCL to achieve this without sacrificing performance. The Sailfish goals are explicitly listed as performance, scalability, agility and extendability, maintenance, and ease of use. The project scored well in these categories during our assessment. The quality priorities for Sailfish roughly match the priorities observed for research software in general. Nguyen-Hoan et al. (2010) surveyed developers to find the following list of qualities, in decreasing order of importance: reliability, functionality, maintainability, availability, performance, flexibility, testability, usability, reusability, traceability, and portability. The

Sailfish list does not list reliability or functionality, but we can safely assume those are implicitly high priorities for any scientific project. In earlier studies Kelly and Sanders (2008) and Carver et al. (2007) highlight how important correctness is for research software.

During our interviews, documentation was noted as playing a significant role in the development process, specifically with on-boarding new developers. A goal of documentation is to lower the entry barrier for these new contributors. The documentation provides information on how to get started, orients the user to artifacts and the source code, and explains how the system works, including the so-called simulation engine and interface. This emphasis on documentation, especially for new developers, is echoed in research software guidelines. Multiple guidelines recommend a document explaining how to contribute to a project, often named CONTRIBUTING (Yehudi, 2021; Brett et al., 2021; Wilson et al., 2016; Thiel, 2020; van Gompel et al., 2016; Orviz et al., 2017; Free/Libre and Software, 2022; Jiménez RC et al., 2017). Tutorials (Thiel, 2020), trouble shooting guides (Orviz et al., 2017; Institute, 2022) and quick start examples (Thiel, 2020; van Gompel et al., 2016) are also recommended. (Smith et al., 2018a) suggests including instructions specifically for on-boarding new developers. For open source software in general (not just research software), (Fogel, 2005) recommends providing tutorial style examples, developer guidelines, demos and screenshots.

8. Developer Pain Points

Based on interviews with 4 developers, this section aims to answer the research questions: i) What are the pain points for developers working on research software projects (RQ7)?; and, ii) How do the pain points of developers from LBM compare to the pain points for research software in general (RQ8)? Below we go through each of the identified pain points and include citations that contrast the LBM experience with observations from researchers in other domains. Potential ways to address the pain points are covered in Section 9. The full interview questions are found in Smith et al. (2021).

(Pinto et al., 2018) lists some pain points that did not come up in our conversations with LBM developers: Cross-platform compatibility, interruptions while coding, scope bloat, lack of user feedback, hard to collaborate on software projects, and aloneness. (Wiese et al., 2019) repeat some of the previous pain points and add the following: dependency management, data handling concerns (like data quality, data management and data privacy), reproducibility, and software scope determination. Although LBM developers did not mention these pain points, we cannot conclude that they are not relevant for LBM software development, since we only interviewed 4 LBM developers for about an hour each.

P1: **Lack of Development Time** A developer of pyLBM noted that their small development team has a lack of time to implement new features. Small development teams are common for LBM software packages (as shown in the measurement table excerpt in Figure 2). Lack of time is also highlighted as a pain point by other research software developers Pinto et al. (2018, 2016); Wiese et al. (2019).

P2: **Lack of Software Development Experience** A lack of software development experience was noted by the developer of TCLB, and others noted a need for improving software engineering education. Many of the team members on their project are domain experts, not computer scientists or software engineers. This same trend is noted by Nguyen-Hoan et al. (2010), which showed only 23% of research software survey respondents having a computing-related background. Similarly, (Nangia and Katz, 2017) show that the majority (54%) of postdocs have not received training in software development. The LBM developer suggesting an increasing role for formal software education matches the trend observed by Pinto et al. (2018), where their replication of a previous study (Hannay et al., 2009), shows a growing interest in formal training (From 13% of respondents in 2009 to 22% in 2018). (Pinto et al., 2018) found that some developers feel there is a mismatch between coding skills and subject-matter skills.

P3: **Lack of Incentive and Funding** The TCLB developer noted a lack of incentives and funding in academia for developing widely used research software. This problem has also been noted by others (Gewaltig and Cannon, 2012; Goble, 2014; Katerbow and Feulner, 2018). Wiese et al. (2019) reported developer pains related to publicity, since historically publishing norms make it difficult to get credit for creating software. As studied

by Howison and Bullard (2016), research software (specifically biology software, but the trend likely applies to other research software domains) is infrequently cited. (Pinto et al., 2018) also mentions the lack of formal reward system for research software.

P4: **Lack of External Support** A concern was raised that there are no organizations helping with the development of good quality software. This concern is not echoed in the literature because there are such organizations, including Better Scientific Software (BSSw), Software Sustainability Institute (Crouch et al., 2013), and Software Carpentry (Wilson and Lumsdaine, 2006; Wilson, 2016). Over time awareness of these groups will grow, so this pain point is likely to disappear in the future for LBM and other research software developers.

P5: **Technology Hurdles** Technology pain points for LBM developers include setting up parallelization and continuous integration. The pain point survey of (Wiese et al., 2019) also highlighted technical-related problems like dependency management, cross-platform compatibility, continuous integration, hardware issues and operating system issues.

P6: **Ensuring Correctness** Difficulties with ensuring correctness were noted by several developers. They alluded to difficulty with testing the correctness of large numbers of features, and challenges with automated automated testing. The TCLB developer commented that the amount of testing data needed was sometimes problematic, since free testing services do not offer adequate facilities for large amounts of data, which means in-house testing solutions are needed. Other research software domains point to the following problems with testing: i) (Pinto et al., 2018) mention the problem of insufficient testing; ii) the survey of (Hannay et al., 2009) shows that more developers think testing is important than the number that believe they have a sufficient understanding of testing concepts; and, iii) (Hannay et al., 2009; Kanewala and Bieman, 2013; Kelly et al., 2011; Wiese et al., 2019) point to the oracle problem, which occurs in research software when we do not have a means to judge the correctness of the calculated solutions. The LBM experience seems to overlap with i and ii, but the LBM developers did not allude to problem iii (the oracle problem) in our conversations. As discussed in Section 9.2, the oracle problem likely did not come up because the LBM developers we interviewed have strategies for verifying their work.

P7: **Usability** Several developers noted that users sometimes try to use incorrect LBM method combinations to solve their problems. Furthermore, some users think that the packages will work out of the box to solve their cases, while in reality CFD knowledge needs to be applied to correctly modify the packages for the new endeavour. Some of the respondents in the survey of (Wiese et al., 2019) also mentioned that users do not always have the expertise required to install or use the software.

P8: **Technical Debt** The developer of ESPResSo said that their source code was written with a specific application in mind, which later caused too much coupling between components in the source code. This results in technical debt Kruchten et al. (2012), which has an impact on future modifiability and reusability. Concern with technical debt is likely why researchers in the survey of (Nguyen-Hoan et al., 2010) rated maintainability as the third most important software quality. More recently the push for sustainable software de Souza et al. (2019) is motivated by the pain that past developers have had with accumulating too much technical debt.

P9: **Quality of Documentation** The importance of documentation for both users and developers was stressed throughout the interviews. However, it was noted several times that a lack of time and funding (P1) has a negative affect on the documentation. Most of the developers are scientific researchers evaluated on the scientific papers that they produce. Writing and updating documentation is something that is done in their free time, if that time arises. Inadequate research software documentation is also mentioned by others (Pinto et al., 2018; Wiese et al., 2019) and the problem also arises with non-research software (Lethbridge et al., 2003). Recommendations on the development of research software state that developers should critically evaluate their own development processes in terms of quality assurance and comply with international standards for software documentation (Katerbow and Feulner, 2018).

9. Lessons from LBM Developers

This section summarizes the best practices from LBM developers that are taken to address the pain points mentioned in Section 8. The main source of information is the qualitative data from developer interviews (Section 2.7). The practices summarized in this section can potentially be emulated by the LBM software packages that do not currently follow them. Moreover, these practices may also provide examples that can be followed by other research software domains.

9.1. Design For Change

To address technical debt (P8), the top LBM developers show how modularization can be used to design research software for future change. Although the advice to modularize research software to handle complexity is common (Wilson et al., 2014; Stewart et al., 2017; Storer, 2017), specific guidelines on how to divide the software into modules is less prevalent. Not every decomposition is a good design for supporting change, as shown by Parnas (1972). A design with low cohesion and high coupling (Ghezzi et al., 2003, p. 48) will make change difficult. Especially in research software, where change is inevitable, designers need to produce a modularization that supports change.

LBM developers highlighted in our interviews cases where their modularizations anticipated future changes. The developer of pyLBM mentioned that the geometries and models of their system had been "decoupled", using abstraction and modularization of the source code, to make it "very easy to add [new] features". The pyLBM design allows for independent changes to the geometry and the model. We also learned that the package pyLBM redeveloped data structures to ease future changes. The developer of TCLB noted that their software package was designed to allow for the addition of some LBM features, but changes to major aspects of the system would be difficult. For example, "implementing a new model will be an easy contribution", but changes to the "Cartesian mesh ... will be a nightmare". The design of TCLB highlights that not every conceivable change needs to be supported, only the likely changes.

As the LBM developers illustrate, design for changes is accomplished by first identifying likely changes, either implicitly or explicitly, and second by hiding each likely change behind a well-defined module interface. Although it is unclear whether the developers are aware of Parnas's work, the approach mirrors his recommendations (Parnas, 1972).

9.2. Circumventing the Oracle Problem and Addressing Reproducibility

The top LBM projects have adopted techniques for software verification that get around the oracle problem (mentioned earlier in Section P6). In the absence of a general test oracle, the following techniques have been adopted by LBM developers:

- Compare the calculated results against manually calculated results in the cases where manual solutions can be determined. Although it is not always feasible to calculate results a priori, developers can find sub-sets of the general problem where answers can be determined. In special cases that allow this, the results can be compared against exact analytical solutions, as done by OpenLB.
- Related to peer review, verify the mathematical foundations of the models to build confidence in the software's theoretical underpinnings.
- Compare the calculated results against pseudo-oracle solutions for verification benchmarks. The developer of ESPResSo mentioned the value of "tests which compare implementations against each other" where the tests "rely on different principles and they've been written by different people". Latt et al. (2021) shows several benchmark tests comparing Palabos solutions to other software solutions, such as solutions calculated using AnSys.
- Use peer review of the code for major changes and additions, as mentioned in Section 7 for ESPResSo.

Additional strategies for getting around the oracle problem are provided in Section 10.

Some LBM developers improve reproducibility via automating the verification tests and making them part of a continuous integration process (mentioned in Section 6). Providing unit tests improves reproducibility because future developers will require a means to verify that they can obtain the same computational results as today. Unit tests are a good way to embed self-diagnostics of reproducibility in the code (Benureau and Rougier, 2017).

9.3. Prioritize Documentation and Usability

Although documentation is difficult (as mentioned under pain point P9), some LBM developers have realized that prioritizing time on documentation provides a justifiable return on investment. For instance, the interviewed developer of ESPResSo noted that all major additions to their package had accompanying changes to artifacts and documentation. They noted that considerable effort was put into their documentation. They further commented that their goal was to lower the entry barrier for new developers, which meant providing considerable developer documentation. This documentation informs developers on how to get started; orients them to the artifacts, source code, system architecture, and build system; and, explains the coupling between the simulation engine and the interface. In addition, ESPResSo developers included API documentation, although they are the only one of the top five packages to do so.

Furthermore, documentation is used by LBM developers to address usability challenges (P7), as follows:

- Explicitly state appropriate fluid dynamics problems that the software is designed to model and explicitly state the limits of the software. This is done in the ESPResSo user guide.
- Provide documentation that details the background theory, or provide a reference to such information. This was done by several top ranked packages, including ESPResSo, Ludwig, LUMA, and OpenLB.
- Identify expected user characteristics. LIMBES, Ludwig, laboetie, and Palabos all did this. Explicit user characteristics are important so the application and documentation can be pitched toward the right user background (Smith et al., 2007).
- Keep all documentation in one location. This was done by all top ranked packages.
- Use documentation generators, like Doxygen, as done (for instance) by OpenLB. More details on the packages that use generators is given in Sections 3.2 and 6.
- Provide an explanation of theoretical principles through the user interface.
- Add guards in the code to test user inputs for compatibility before proceeding with the calculations.
- Include documentation on frequently asked questions (Section 5).

10. Recommendations for Future Practices

In this section we provide recommendations to address the pain points from Section 8 to answer RQ10. Our recommendations are not lists of criticisms what should have been done in the past, or what should be done now; they are just suggestions for consideration in the future. We will not be repeating ideas here that have been discussed in previous section, such as continuous integration, documentation of APIs, etc. Our aim is to mention ideas that are at least somewhat beyond conventional best practices. The ideas listed have the potential to become best practices in the medium to long-term. The ideas are provided roughly in the order of increasing novelty.

10.1. Employ Linters

With the exception of Thiel (2020), the research software guidelines that we consulted do not mention linters. However, we feel linters have the potential to improve code quality at a relatively low cost. A linter is a tool that statically analyzes code to find programming errors, stylistic inconsistencies and suspicious constructs (Wikipedia, 2022). Linters can be used to spot check code files, or even better as part of a continuous integration system.

A sample of the potential benefits of linters include: finding memory leaks, finding potential bugs, standardizing code with respect to formatting, improving performance, removing silly errors before code reviews, and catching potential security issues (SourceLevel, 2022). Linters are available for most popular programming languages. For instance Python has the options of PyLint, flake8 and Black (Zadka, 2018).

Linters can address the pain points for LBM developers. For instance, a linter can increase the amount of development time (P1) by decreasing the number of mundane mistakes programmers have to catch. Since the linter can include rules that capture the wisdom of senior programmers, it can help newer developers avoid common mistakes (P2). Although there do not appear to be many linters for checking parallel programming mistakes (P5), Parallel Lint

is an option for OpenMP. If a linter is used consistently this can guard against some technical debt (P8). Although linters are thought of as tools for code analysis, similar ideas can be applied to documentation to ensure that at least the basic documentation rules are followed. The use of tools to check documentation is a partial explanation for the relatively higher quality of statistical tools that are part of the Comprehensive R Archive Network (CRAN) (Smith et al., 2018d).

10.2. Conduct Rigorous Peer Reviews

Peer review is already a strategy adopted by some LBM developers to improve confidence in their software (Section 9.2), but the benefits of peer review can be improved by making its application more rigorous. Jones (2008) shows that rigorous inspection finds 60-65% of latent defects on average, and often tops 85% in defect removal efficiency. By way of comparison, most forms of testing average between 30 and 35% for defect removal efficiency (Ebert and Jones, 2009; Jones, 2008). A formal code inspection involves asking the reviewer to either follow a review checklist (check consistency of variable names, look for terminating loops, etc.), or perform specific review tasks (like summarize the purpose of the code, create a data dictionary for the module assigned, cross reference the code to the technical manual, etc.) The task based inspection approach has been effectively used for research software, as described by Kelly and Shepard (2000). Task based inspection is an ideal fit with an issue tracking system, like GitHub. The review tasks can be issues, so that they can be easily assigned, monitored and recorded. Other potential issues for the tracker include assigning junior team members to test installation instructions and getting-started tutorials.

Rigorous peer review addresses the same pain points as linters (Section 10.1): P1, P2, P5), P8 and P6. Peer review addresses these pain points by efficiently searching for defects and problems.

10.3. Write and Submit More Papers on Software

To address the pain point of a lack of funding (P3), LBM developers should watch for new opportunities to publish their research software source code. For instance, the Journal of Open Source Software (JOSS) reviews and publishes articles that credit the scholarship contained in the software itself (Smith et al., 2018c). (Smith et al., 2016a) presents a set of software citation principles that may encourage broad adoption of a consistent policy for software citation across disciplines and venues. (Chue Hong et al., 2019) provides a software citation checklist for developers to make a release of their software citable. Further guidance on software citation is provided by Katz et al. (2021). Another option is to use GitHub for citation, since it is relatively easy to add metadata to repositories and to generate citations for those repositories Smith (2022a).

10.4. Grow the Number of Contributors

In our interviews with developers we heard they would like to have additional software contributors. More developers would help with addressing the lack of development time pain point (P1). In investigating advice on increasing the number of contributors, we found that the advice usually starts with following best practices, including providing artifacts like those discussed in Section 5 such as a contributor guidelines, a clear code of conduct and high quality code and documentation. This advice is logical, but in our assessment of LBM software we found examples of projects that already follow many best practices, but still have small development teams. Attracting developers apparently requires more than just building a good product.

Some potential additional ideas for growing the number of contributors are as follows:

- Clearly identify issues that are an appropriate starting points for new developers (Garcia, 2016; Jalan, 2016; Proffitt, 2017).
- Given that in most projects developers were once users (McQuaid, 2018), recruit future developers from the current set of users.
- Create templates for pull requests and issues (Jalan, 2016).
- Welcome all kinds of contributions, not just code. Non-code contributions include documentation, fixing typos, issue reporting and test cases (Jalan, 2016; Proffitt, 2017).

- Reward and recognize new contributors, via small rewards like stickers or shirts, or even just a simple shoutout or mention in a blogpost or on social media (Jalan, 2016; Proffitt, 2017).
- Recognize that open source is more about people than it will ever be about code (Jalan, 2016).
- Look beyond just recruiting online and seek new developers at conferences or other user meet ups (Garcia, 2016).
- Follow the advice of Kuchner (2012) to adapt ideas from marketing to promote science. Specific ideas that could be applied to open source projects include the following:
  - Storytelling to motivate interest in the project, where the story is a sequence of events and pauses for reflection (Kuchner, 2012, p. 21–22).
  - Invest effort in building relationships with users and with potential future developers.
  - Brand your project with what makes it unique and special.
- Consider combining existing projects to pool collective resources, and reduce competition between different open source projects.

10.5. Further Address the Oracle Problem

To address the pain point of ensuring correctness (P6), Section 9.2 outlined techniques that LBM developers use to circumvent the oracle problem for testing. Additional techniques to build system tests are as follows (Smith, 2016):

- Create test cases by assuming a solution and using this solution to calculate the necessary inputs. For instance, for a linear solver, if A and x are assumed, b can be calculated as $b = Ax$. Following this, $Ax^* = b$ can be solved and then x and x^*, which should theoretically be equal, can be compared. In the case of solving Partial Differential Equations (PDEs), this approach is called the Method of Manufactured Solutions (Roache, 1998).
- Use interval arithmetic to build test cases. For testing purposes, the slow, but guaranteed correct, interval arithmetic (Hickey et al., 2001) can be employed. The faster floating point algorithm can then be verified by ensuring that the calculated answers lie within the guaranteed bounds.
- Include convergence studies. The discretization used in the numerical algorithm should be decreased (usually halved) and the change in the solution assessed. This is very likely already done by LBM developers, but it did not come up during our interviews.
- Use *metamorphic* testing, where the program is checked to see if it satisfies a set of metamorphic relations, where a metamorphic relation is a relationship between multiple inputs and output pairs (Kanewala and Lundgren, 2016). An example could be the relationship that the output flow rate increases as the input flow rate increases.

10.6. Augment Theory Manual to Include Requirements Information

As discussed in Section 5, requirements specifications are rarely written for research software. Although a full requirements specification has advantages (Smith et al., 2007; Smith and Lai, 2005), given the lack of development time (P1), we recommend the intermediate step of augmenting the existing theory manuals with some requirements information. Table 7 shows that theory documentation is common. We observed 17 of 24 packages had at least some theory documentation. Templates for scientific requirements, like Smith et al. (2007); Smith and Lai (2005), show theory is a significant part of the requirements documentation. With the addition of some extra information, the theory documents can be transformed into requirements specification. The key extra information includes explicitly stating user characteristics, explicitly stating how the user interacts with the software in terms of input data requirements, and listing likely and unlikely changes. Explicit statements about likely changes are invaluable in the design stage, since they provide developers guidance on how general the software needs to be. For those wishing to maximize the value of a requirement specification, information could be added like prioritizing the nonfunctional requirements (to show the relative importance between qualities like portability, reliability and performance) and traceability information (to

show the consequences of changes to the assumptions). Spending time to incorporate additional information in the theory manuals should help address the quality of existing documentation (P9) and improve the software's usability (P7).

10.7. Improve Re-runnability, Repeatability, Reproducibility and Replicability

Ensuring correctness was identified as a pain point by developers (P6). Inspired by Benureau and Rougier (2017), we make recommendations to ensure that correctness can be maintained into the future by prioritizing 4 increasingly sophisticated levels of replication.

Re-runnability means that today's code can be run again in the future when needed. Re-runnability becomes more difficult as code ages and the software and hardware infrastructure around it changes, like changes to compilers, GPUs, libraries etc. To be re-runnable on other researchers' computers, a code should provide a detailed description of the execution environment in which it is executable. Besides recording details, virtual machines, docker and environment managers, like Conda, can be used to improve re-runnability.

Repeatability means producing the same output over successive runs of a program. As was pointed out during our developer interviews, this can be difficult for LBM software because of dependence on probability distributions. However, deterministic output is important for testing purposes and for debugging problematic code. For LBM software repeatability will likely require running a serial version of the code with full control of the initialization of the pseudo-random number generators.

Reproducibility Once re-runnability and repeatability are achieved, the next step is reproducibility. Current work by LBM developers on reproducibility is discussed in Section 9.2. The proposed goal for the future is for all results to be reproducible, which means documenting dependencies, platforms, parameter values and input files. The data and scripts behind the graphs should be published (Benureau and Rougier, 2017). Some tools and techniques for computational reproducibility are summarized in Piccolo and Frampton (2016).

Replicability means that the documentation provided for the software, not the software itself, is sufficient for an independent third party to reproduce the computational results. Reproducibility does not help us if our trust in the original code is in doubt. What if we need to reproduce the results not starting from the code, but starting from the original theory? Unfortunately, multiple examples exist (Crick et al., 2014; Ionescu and Jansson, 2012) where results from research software could not be independently reproduced, due to a need for local knowledge missing from published documents. Examples of missing knowledge includes missing assumptions, derivations, and undocumented modifications to the code. In the future, we wish for it to be easier to independently replicate the work of others, starting from their theory documentation, augmented with requirements information as discussed above, without needing to rely on their code.

10.8. Generate All Things

We propose automatically generating LBM code and its documentation, using a Domain Specific Modelling (DSM) approach. DSM means creating a knowledge base of models for physics, computing, mathematics, documentation and certification and then writing explicit "recipes" that weave together this knowledge to generate the desired code, documentation, test cases, inspection reports and build scripts. This definition of DSM is more general than usual; in this recommendation DSM implies generation of all software artifacts, not just the code. DSM moves development to a higher level of abstraction; domain experts can work without concern for low-level implementation details. Using DSM, code and documentation can be optimally generated, redundancy is eliminated, the likelihood of errors is reduced and maintenance is automated. Moreover, a generative approach facilitates the inevitable exploration of LBM modelling assumptions.

DSM provide a transformative technology for documentation, design and verification (Johanson and Hasselbring, 2018; Smith, 2018). DSM allows scientists to focus on their science, not software. A generative approach removes the maintenance nightmare of documentation duplicates and near duplicates (Luciv et al., 2018), since knowledge is only captured once and transformed as needed. Code generation has previously been applied to improve research software. For instance, ATLAS (Automatically Tuned Linear Algebra Software) (Whaley et al., 2001) and Blitz++ (Veldhuizen, 1998) produces efficient and portable linear algebra software. Spiral (Püschel et al., 2001) uses software/hardware

generation for digital signal processing. Carette and Kiselyov (2011) shows how to generate a family of efficient, type-safe Gaussian elimination algorithms. FEniCS (Finite Element and Computational Software) uses code generation when solving differential equations (Logg et al., 2012). Ober et al. (2018) and Matkerim et al. (2013) apply DSM to High Performance Computing (HPC), using UML (Unified Modelling Language) for their domain models. Unlike previous DSM work, the current recommendation focuses on generating all software artifacts (requirements, design, etc.), not just code. Initial work on this "generate all things" approach is discussed in (Szymczak et al., 2016) with a prototype available (Carette et al., 2021).

A DSM approach addresses multiple pain points. For instance, once the infrastructure is in place, a DSM can decrease the amount of development time (P1) by automation. Since a DSM approach allows scientists to focus on their science, rather than software, the lack of software development experience (P2) will be less of an issue. The DSM approach can capture computing knowledge to mitigate the technology related pain points (P5). For ensuring correctness (P6) a generative approach can aim to be correct by construction. In cases where mistakes are made, DSM has the advantage that the mistake is propagated throughout the generated artifacts, which greatly increases the chance that the mistake is noted. Technical debt (P8) is not a concern with a DSM approach since the recipes that are used for generation are written at a high level and are thus relatively easy to change. With respect to pain points on usability (P7) and documentation (P9), the generative approach we are proposing addresses these directly, since documentation is a primary concern, rather than an afterthought. and quality of documentation

11. Threats To Validity

The threats to validity can be categorized into methodology and data collection issues. With respect to our methodology, the measurement template may not be detailed enough to accurately capture some qualities. For example, there are only two measures of surface robustness. The measurement of robustness could be expanded, as it currently only measures unexpected input. Other intentional faults could be introduced to assess robustness, but this would require a larger investment of measurement time. Similarly, reusability (Section 3.8) is assessed via the number of code files and LOC per file. While this measure is indicative of modularity, it is possible that some packages have many files, with few LOC, but the files do not contain source code that is easily reusable. The files may be poorly formatted, or the source code may be vague and have ambiguous identifiers. Maintainability may also be incorrectly estimated, since it is assumed to be improved by a higher ratio of comments to source code (Section 3.7), but this is not necessarily true. Furthermore, the measurement of understandability (Section 3.9) relies on 10 random source code files. By random chance, the 10 files that were chosen may not be a good representation of the understandability of that package.

In our methodology one individual measures each package and the beginning and ending of the measurement phase is separated by several months. There is a threat that the measurement may include unconscious biases, or the reviewers judgement may drift over time. In previous work (Smith et al., 2016b), we demonstrated that the measurement process is reasonably reproducible by having 5 products graded by a second reviewer. The ranking via this independent review was almost identical to the original ranking. As long as each grader uses consistent definitions, the relative comparisons in the AHP results will be consistent between graders.

Regarding data collection, a risk to the validity of this assessment is missing or incorrect data. Some data may not have been measured due to technology issues like broken links. This issue arose with the measurement of Palabos, which had a broken link to its user manual, as noted in Section 5. Some pertinent data may not have been specified in public artifacts, or may be obscure within an artifact or web-page. For instance, the use of unit testing and continuous integration was mentioned in the artifacts of only three (ESPResSo, Ludwig, Musubi) packages (Section 3.2). However, interviews suggested a more frequent use of both unit testing and continuous integration. For example, OpenLB, pyLBM, and TCLB use such methods during development despite this not being explicit from an analysis of the material available online.

Furthermore, design documentation was measured to be a "less common" artifact in this assessment, but it is probable that such documentation is part of all LBM packages. After all, developing research software is not a trivial endeavour. It is likely that many packages have such documentation but did not make it public, and due to this the measured data may not be a true reflection of software package quality.

Another data collection concern is the gap between measurement of Musubi and the measurement of all other packages (Section 2.3). Unfortunately we were not aware of the existence of Musubi until after the primary data

collection period was completed. This gap has a secondary effect because it caused a mismatch between the time of manual and automated data collection (Section 4). With time and resource constraints we could not redo the manual measurement, but we did update the automated measures (like the number of files, LOC, etc.) when Musubi was added.

Our comparison between our ranking and the community's ranking (Section 4) may not be valid. Using stars to estimate the community's ranking is not necessarily accurate. Moreover, our overall AHP ranking makes the unrealistic assumption of equal weighting between qualities (Section 3.11).

A final threat to validity is conceptual. Our work assumes that following best practices leads to improved software quality. As the advice in software engineering guidelines show, we are not alone in this assumption. Ideally software qualities should be measured directly and then the impact of best practices on those qualities should be assessed. However, researchers in this domain always face the challenge that many qualities, like maintainability, reusability and verifiability, are difficult to measure directly. Therefore, we follow the usual approach of assuming that following procedures and adhering to standards yields a higher-quality product (van Vliet, 2000, p. 112).

12. Conclusion

To help users select the right LBM package for their needs, and to potentially provide direction for future LBM software development, our goal was to analyze the current state of LBM software development. We aimed to determine how well best practices are followed and to understand current developer pain points. We compared the state of the practice for LBM software to the development practices recommended and employed for research software in general. To improve the quality of LBM software we looked for examples where developers have been successful so that we can share these with the LBM community, and with the broader research software community. We also aimed to identify potential approaches for mitigating the identified pain points in the future.

Our analysis points to areas where some LBM software packages fall short of current best practices. This is not intended to be a criticism of any existing packages, especially since in practice not every project needs to achieve the highest possible quality. However, rather than delve into the nuances of which software can justify compromising which practice we conducted our comparison under the ideal assumption that every project can have sufficient resources to match best practices.

Our methodology involved the following steps: i) Identify a list of 45 candidate software packages; ii) Filter the list to a length of 24 packages; iii) Collect repository related data on each software package, like number of stars, number of open issues, number of lines of code; iv) Fill in the measurement template (the template consists of 108 questions to assess 9 qualities (installability, correctness/verifiability, reliability, robustness, usability, maintainability, reusability, understandability, and visibility/transparency). Filling in the template requires installing the software on a VM, running simple tests (like completing the getting started instructions (if present)), and searching the code, documentation and test files; v) Rank the software using the Analytic Hierarchy Process (AHP); vi) Interview developers (the interview consists of 20 questions and takes about an hour); and, vii) Conduct a domain analysis. The collected data was analyzed by: a) comparing the ranking by best practices against the ranking by popularity; b) comparing artifacts, tools and processes to current research software development guidelines; and, c) exploring pain points. Our ranking was roughly consistent with ranking the repositories by the number of GitHub stars, although the number one project by star count (Sailfish) is ranked 9th by our methodology.

Current LBM software provides many examples of following best practices, to the benefit of users and developers. This is illustrated by Table 9, which summarizes the top 5 performers for each of the identified qualities and for the overall quality. Taking the union of all entries in the table, we have 16 top performers: ESPResSo, MechSys, OpenLB, DL_MESO, Sailfish, Ludwig, Musubi, pyLBM, waLBerla, laboetie, lettuce, ESPResSo++, LUMA, MP-LABS, Palabos, and HemeLB. Therefore, two thirds (67%) of the 24 measured packages appear in the top 5 for at least one quality. Of these entries almost all of them are classified as alive, except for three packages: laboetie, MP-LABS, and HemeLB. This suggests that alive packages are in a healthy state with respect to adopting best practices.

Further evidence of the healthy state of the practice can be seen by comparing the observed software artifacts versus those recommended by software engineering guidelines. As Section 5 shows, the majority of LBM generated artifacts correspond to general recommendations from research software developers. LBM software even seems to be ahead of best practices with an example of a video user guide for one of the packages. Comparing LBM with respect

Quality	Ranked 1st	Ranked 2nd	Ranked 3rd	Ranked 4th	Ranked 5th
Installability	ESPResSo	MechSys	OpenLB	DL_MESO (LBE)	Sailfish
Surface Correctness and Verifiability	Ludwig	Musubi	pyLBM	OpenLB	waLBerla
Surface Reliability	DL_MESO (LBE)	laboetie*	lettuce	MechSys	Musubi
Surface Robustness	DL_MESO (LBE)	ESPResSo++	lettuce	LUMA	OpenLB
Surface Usability	laboetie*	ESPResSo	OpenLB	Ludwig	Sailfish
Maintainability	Ludwig	MP-LABS*	ESPResSo++	ESPResSo	Palabos
Reusability	ESPResSo	waLBerla	ESPResSo++	Sailfish	HemeLB*
Surface Understandability	LUMA	Palabos	pyLBM	DL_MESO (LBE)	MechSys
Visibility and Transparency	Ludwig	Palabos	LUMA	ESPResSo	MP-LABS*
Overall Quality	ESPResSo	Ludwig	Palabos	OpenLB	LUMA

Table 9: Top performers for each quality (all packages were alive at the time of measurement, except for those marked by an asterisk (*))

to tool usage (Section 6) shows that LBM is keeping pace with other research software, with 67% of the surveyed tools using version control. The state of the practice for the software development process of LBM software matches the quasi-agile process generally adopted for research software (Section 7). The best practice process of peer review is employed by some LBM developers.

Although the state of the practice for LBM software is healthy, we have noted areas for potential improvement in the adoption of best practices, as follows:

- Increase the utilization of continuous integration.
- Increase the amount of API documentation.
- Add roadmap documentation showing what is planned for the future.
- Add a code of conduct to explicitly show how developers should treat one another.
- Increase the use and enforcement of programming style guides.
- Use checklists to ensure best practices are followed by developers.
- Include uninstall instructions.
- Explicitly document the software's motivation and design principles.

To advance the state of the practice beyond current best practices toward future best practices, we interviewed 4 developers to learn about their pain points (Section 8). The identified pain points include lack of development time, lack of software development experience, lack of incentive and funding, lack of external support, technology hurdles, ensuring correctness, usability, technical debt and documentation quality. We found (Section 9 that developers are currently creating practices to address these pain points, including: designing for change, circumventing the oracle testing problem, and prioritizing documentation and usability. To further plan for future best practices, we identified eight potential directions (Section 10), in order of increasing novelty:

1. Employ linters.
2. Conduct rigorous peer reviews.
3. Write and submit papers on software.
4. Grow the number of contributors.
5. Further address the oracle problem for testing.
6. Augment theory manuals to include requirements information.
7. Improve re-runnability, repeatability, reproducibility and replicability.
8. Generate all things.

Future work can be done to address the threats to validity mentioned in Section 11. For instance, the measures listed in our measurement template could be broadened to better capture some qualities. For example, usability experiments and performance benchmarks could be incorporated into the assessment. Section 11 also mentions that some pertinent information on continuous integration and unit testing was not specified in public artifacts. Adding further questions to the interview guide regarding the measures that are on the measurement template could reduce instances of incomplete data. This additional interview data could be analyzed and incorporated into the AHP ranking, ensuring quality designations more accurate represent the true quality of the software packages. In the future we could also expand the interview questions to go beyond reproducibility to also discuss replicability. Furthermore, in future interviews we would like to clarify the distinction between usability and understandability.

Our domain analysis (Section 2.8 was relatively shallow. In the future we would like to dig more deeply into this analysis. We would like to produce tables of commonalities, variabilities, and parameters of variation for the family of LBM solvers. We could follow the template from (Smith et al., 2008), or adopt the analysis techniques from (Weiss, 1998). Smith and Chen (2004) and Smith et al. (2017) are examples of a commonality analysis for a family of mesh generating software and a family of material models, respectively.

Acknowledgements

We would like to thank Ao Dong, Oluwaseun Owojaiye, Mohammad Ali Daeian and Reza Sadeghi for fruitful discussions on topics relevant to this paper.

References

2017. ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary. *ISO/IEC/IEEE 24765:2017(E)* (2017), 1–541. https://doi.org/10.1109/IEEESTD.2017.8016712

Karen S. Ackroyd, Steve H. Kinder, Geoff R. Mant, Mike C. Miller, Christine A. Ramsdale, and Paul C. Stephenson. 2008. Scientific Software Development at a Research Facility. *IEEE Software* 25, 4 (July/August 2008), 44–51.

Yasmin AlNoamany and John A. Borghi. 2018. Towards computational reproducibility: researcher perspectives on the use and sharing of software. *PeerJ Computer Science* 4, e163 (September 2018), 1–25.

Shadab Anwar and Michael C. Sukop. 2009. Regional scale transient groundwater flow modeling using Lattice Boltzmann methods. *Computers & Mathematics with Applications* 58, 5 (2009), 1015–1023. https://doi.org/10.1016/j.camwa.2009.02.025 Mesoscopic Methods in Engineering and Science.

Yuanxun Bill Bao and Justin Meskas. 2011. Lattice Boltzmann method for fluid simulations. *Department of Mathematics, Courant Institute of Mathematical Sciences, New York University* (2011), 44.

Martin Bauer, Sebastian Eibl, Christian Godenschwager, Nils Kohl, Michael Kuron, Christoph Rettinger, Florian Schornbaum, Christoph Schwarzmeier, Dominik Thönnes, Harald Köstler, et al. 2021a. waLBerla: A block-structured high-performance framework for multiphysics simulations. *Computers & Mathematics with Applications* 81 (2021), 478–501.

Martin Bauer, Harald Köstler, and Ulrich Rüde. 2021b. lbmpy: Automatic code generation for efficient parallel lattice Boltzmann methods. *Journal of Computational Science* 49 (2021), 101269.

Mario Christopher Bedrunka, Dominik Wilde, Martin Kliemank, Dirk Reith, Holger Foysi, and Andreas Krämer. 2021. Lettuce: PyTorch-based Lattice Boltzmann Framework. In *International Conference on High Performance Computing*. Springer, 40–55.

F. Benureau and N. Rougier. 2017. Re-run, Repeat, Reproduce, Reuse, Replicate: Transforming Code into Scientific Contributions. *ArXiv e-prints* (Aug. 2017). arXiv:1708.08205 [cs.GL]

Barry W Boehm. 2007. *Software engineering: Barry W. Boehm's lifetime contributions to software development, management, and research*. Vol. 69. John Wiley & Sons.

Alys Brett, James Cook, Peter Fox, Ian Hinder, John Nonweiler, Richard Reeve, and Robert Turner. 2021. Scottish Covid-19 Response Consortium. https://github.com/ScottishCovidResponse/modelling-software-checklist/blob/main/software-checklist.md.

Titus Brown. 2015. Notes from"How to grow a sustainable software development process (for scientific software)". http://ivory.idyll.org/blog/2015-growing-sustainable-software-development-process.html.

Jacques Carette and Oleg Kiselyov. 2011. Multi-stage programming with Functors and Monads: Eliminating abstraction overhead from generic code. *Sci. Comput. Program.* 76, 5 (2011), 349–375.

Jacques Carette, Spencer Smith, Jason Balaci, Anthony Hunt, Ting-Yu Wu, Samuel Crawford, Dong Chen, Dan Szymczak, Brooks MacLachlan, Dan Scime, and Maryyam Niazi. 2021. *Drasil*. https://github.com/JacquesCarette/Drasil/tree/v0.1-alpha

Jeffrey C. Carver, Richard P. Kendall, Susan E. Squires, and Douglass E. Post. 2007. Software Development Environments for Scientific and Engineering Software: A Series of Case Studies. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*. IEEE Computer Society, Washington, DC, USA, 550–559. https://doi.org/10.1109/ICSE.2007.77

Shiyi Chen and Gary D Doolen. 1998. Lattice Boltzmann method for fluid flows. *Annual review of fluid mechanics* 30, 1 (1998), 329–364.

Z Chen, C Shu, LM Yang, X Zhao, and NY Liu. 2021. Phase-field-simplified lattice Boltzmann method for modeling solid-liquid phase change. *Physical Review E* 103, 2 (2021), 023308.

Neil P. Chue Hong, Alice Allen, Gonzalez-Beltran, Anita de Waard, Arfon M. Smith, Carly Robinson, Catherine Jones, Daina Bouquin, Daniel S. Katz, David Kennedy, Gerry Ryder, Jessica Hausman, Lorraine Hwang, Matthew B. Jones, Melissa Harrison, Mercè Crosas, Mingfang Wu, Peter Löwe, Robert Haines, Scott Edmunds, Shelley Stall, Sowmya Swaminathan, Stephan Druskat, Tom Crick, Tom Morrell, and Tom Pollard. 2019. Software Citation Checklist for Developers. https://doi.org/10.5281/zenodo.3482769

Tom Crick, Benjamin A. Hall, and Samin Ishtiaq. 2014. "Can I Implement Your Algorithm?": A Model for Reproducible Research Software. *CoRR* abs/1407.5981 (2014). http://arxiv.org/abs/1407.5981

S. Crouch, N. C. Hong, S. Hettrick, M. Jackson, A. Pawlik, S. Sufi, L. Carr, D. De Roure, C. Goble, and M. Parsons. 2013. The Software Sustainability Institute: Changing Research Software Attitudes and Practices. *Computing in Science Engineering* 15, 6 (Nov 2013), 74–80. https://doi.org/10.1109/MCSE.2013.133

Mario Rosado de Souza, Robert Haines, Markel Vigo, and Caroline Jay. 2019. What Makes Research Software Sustainable? An Interview Study With Research Software Engineers. *CoRR* abs/1903.06039 (2019). arXiv:1903.06039 http://arxiv.org/abs/1903.06039

Jean-Christophe Desplat, Ignacio Pagonabarraga, and Peter Bladon. 2001. LUDWIG: A parallel Lattice-Boltzmann code for complex fluids. *Computer Physics Communications* 134, 3 (2001), 273–290.

Ao Dong. 2021. *Assessing the State of the Practice for Medical Imaging Software*. Master's thesis. McMaster University, Hamilton, ON, Canada.

Steve M. Easterbrook and Timothy C. Johns. 2009. Engineering the Software for Understanding Climate Change. *Comuting in Science & Engineering* 11, 6 (November/December 2009), 65–74. https://doi.org/10.1109/MCSE.2009.193

C. Ebert and C. Jones. 2009. Embedded Software: Facts, Figures, and Future. *Computer* 42, 4 (April 2009), 42–52. https://doi.org/10.1109/MC.2009.118

ESA. February 1991. *ESA Software Engineering Standards, PSS-05-0 Issue 2*. Technical Report. European Space Agency.

Karl Fogel. 2005. *Producing Open Source Software: How to Run a Successful Free Software Project*. O'Reilly Media, Inc.

Free/Libre and Open Source Software. 2022. FLOSS Best Practices Criteria (Passing Badge). https://bestpractices.coreinfrastructure.org/en/criteria/0.

SA Galindo-Torres. 2013. A coupled Discrete Element Lattice Boltzmann Method for the simulation of fluid–solid interaction with particles of general shapes. *Computer Methods in Applied Mechanics and Engineering* 265 (2013), 107–119.

Davood Domairry Ganji and Sayyid Habibollah Hashemi Kachapi. 2015. *Application of nonlinear systems in nanomechanics and nanofluids: analytical methods and applications*. William Andrew.

Jeremy Garcia. 2016. How do you get programmers to join your project? https://opensource.com/business/16/9/how-to-get-programmers.

Lauraine Genota. 2018. Why Generation Z Learners Prefer YouTube Lessons Over Printed Books. Education Week, https://www.edweek.org/teaching-learning/why-generation-z-learners-prefer-youtube-lessons-over-printed-books/2018/09.

Marc-Oliver Gewaltig and Robert Cannon. 2012. Quality and sustainability of software tools in neuroscience. *Cornell University Library* (2012), 1–20.

Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. 2003. *Fundamentals of Software Engineering* (2nd ed.). Prentice Hall, Upper Saddle River, NJ, USA.

Ray Givler. 2020. A Checklist of Basic Software Engineering Practices for Data Analysts and Data Scientists. https://www.linkedin.com/pulse/checklist-basic-software-engineering-practices-data-analysts-givler/?articleId=6681691007303630849.

Carole Goble. 2014. Better Software, Better Research. *IEEE Internet Computing* 18, 5 (2014), 4–8. https://doi.org/10.1109/MIC.2014.88

Benjamin Graille and Loïc Gouarin. 2017. pylbm Documentation. (2017).

Alan Gray and Kevin Stratford. 2013. Ludwig: multiple GPUs for a complex fluid lattice Boltzmann application. *Designing Scientific Applications on GPUs. Chapman & Hall/CRC Numerical Analysis and Scientific Computing Series, Taylor & Francis* (2013).

Jonathan D Halverson, Thomas Brandes, Olaf Lenz, Axel Arnold, Staš Bevc, Vitaliy Starchenko, Kurt Kremer, Torsten Stuehn, and Dirk Reith. 2013. ESPResSo++: A modern multiscale simulation package for soft matter systems. *Computer Physics Communications* 184, 4 (2013), 1129–1149.

Jo Erskine Hannay, Carolyn MacLeod, Janice Singer, Hans Petter Langtangen, Dietmar Pfahl, and Greg Wilson. 2009. How Do Scientists Develop and Use Scientific Software?. In *Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering (SECSE '09)*. IEEE Computer Society, Washington, DC, USA, 1–8. https://doi.org/10.1109/SECSE.2009.5069155

Adrian RG Harwood, Joseph O'Connor, Jonathan Sanchez Muñoz, Marta Camps Santasmasas, and Alistair J Revell. 2018. LUMA: A many-core, fluid–structure interaction solver based on the lattice-Boltzmann method. *SoftwareX* 7 (2018), 88–94.

Manuel Hasert, Kannan Masilamani, Simon Zimny, Harald Klimach, Jiaxing Qi, Jörg Bernsdorf, and Sabine Roller. 2014. Complex fluid simulations with the parallel tree-based lattice Boltzmann solver Musubi. *Journal of Computational Science* 5, 5 (2014), 784–794.

Dustin Heaton and Jeffrey C. Carver. 2015. Claims About the Use of Software Engineering Practices in Science. *Inf. Softw. Technol.* 67, C (Nov. 2015), 207–219. https://doi.org/10.1016/j.infsof.2015.07.011

Michael A. Heroux and David E. Bernholdt. 2018. Better (Small) Scientific Software Teams, tutorial in Argonne Training Program on Extreme-Scale Computing (ATPESC). https://press3.mcs.anl.gov//atpesc/files/2018/08/ATPESC_2018_Track-6_3_8-8_1030am_Bernholdt-Better_Scientific_Software_Teams.pdf. https://doi.org/articles/journal_contribution/ATPESC_Software_Productivity_03_Better_Small_Scientific_Software_Teams/6941438

Michael A. Heroux, James M. Bieman, and Robert T. Heaphy. 2008. Trilinos Developers Guide Part II: ASC Softwar Quality Engineering Practices Version 2.0. https://faculty.csbsju.edu/mheroux/fall2012_csci330/TrilinosDevGuide2.pdf.

Vincent Heuveline and Mathias J Krause. 2010. OpenLB: towards an efficient parallel open source library for lattice Boltzmann fluid flow simulations. In *International Workshop on State-of-the-Art in Scientific and Parallel Computing. PARA*, Vol. 9. 570.

Vincent Heuveline, Mathias J Krause, and Jonas Latt. 2009. Towards a hybrid parallelization of lattice Boltzmann methods. *Computers & Mathematics with Applications* 58, 5 (2009), 1071–1080.

Timothy Hickey, Qun Ju, and Maarten H. Van Emden. 2001. Interval Arithmetic: From Principles to Implementation. *J. ACM* 48, 5 (Sept. 2001), 1038–1068. https://doi.org/10.1145/502102.502106

James Howison and Julia Bullard. 2016. Software in the Scientific Literature: Problems with Seeing, Finding, and Using Software Mentioned in the Biology Literature. *J. Assoc. Inf. Sci. Technol.* 67, 9 (sep 2016), 2137–2155. https://doi.org/10.1002/asi.23538

IEEE. 1991. *IEEE Standard Glossary of Software Engineering Terminology*. Standard. IEEE.

IEEE. 1998. Recommended Practice for Software Requirements Specifications. *IEEE Std 830-1998* (Oct. 1998), 1–40. https://doi.org/10.1109/IEEESTD.1998.88286

Software Sustainability Institute. 2022. Online sustainability evaluation. https://www.software.ac.uk/resources/online-sustainability-evaluation.

Cezar Ionescu and Patrik Jansson. 2012. Dependently-Typed Programming in Scientific Computing — Examples from Economic Modelling. In *Revised Selected Papers of the 24th International Symposium on Implementation and Application of Functional Languages (Lecture Notes in Computer Science, Vol. 8241)*. Springer International Publishing, 140–156. https://doi.org/10.1007/978-3-642-41582-1_9

ISO/IEC. 2001. *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC.

ISO/IEC. 2011. *Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models*. Standard. International Organization for Standardization.

Shubhkesha Jalan. 2016. How to attract new contributors to your open source project. https://www.freecodecamp.org/news/how-to-attract-new-contributors-to-your-open-source-project-46f8b791d787/.

Michal Januszewski and Marcin Kostur. 2014. Sailfish: A flexible multi-GPU implementation of the lattice Boltzmann method. *Computer Physics Communications* 185, 9 (2014), 2350–2368.

Kuzak M Jiménez RC, Alhamdoosh M, and et al. 2017. Four simple recommendations to encourage best practices in research software [version 1; peer review: 3 approved]. *F1000Research* 6, 876 (2017). https://doi.org/10.12688/f1000research.11407.1

Arne N. Johanson and Wilhelm Hasselbring. 2018. Software Engineering for Computational Science: Past, Present, Future. *Computing in Science & Engineering* Accepted (2018), 1–31.

Capers Jones. 2008. Measuring Defect Potentials and Defect Removal Efficiency. *Crosstalk, The Journal of Defense Software Engineering* 21, 6 (June 2008), 11–13.

Panagiotis Kalagiakos. 2003. The Non-Technical Factors of Reusability. In *Proceedings of the 29th Conference on EUROMICRO*. IEEE Computer Society, 124.

U. Kanewala and J. M. Bieman. 2013. Techniques for testing scientific programs without an oracle. In *Software Engineering for Computational Science and Engineering (SE-CSE), 2013 5th International Workshop on*. 48–57. https://doi.org/10.1109/SECSE.2013.6615099

Upulee Kanewala and Anders Lundgren. 2016. Automated Metamorphic Testing of Scientific Software. In *Software Engineering for Science*, Jeffrey C. Carver, Neil Chue Hong, and George Thiruvathukal (Eds.). Taylor & Francis, Chapter Examples of the Application of Traditional Software Engineering Practices to Science, 151–174.

Matthias Katerbow and Georg Feulner. 2018. Recommendations on the development, use and provision of Research Software. https://doi.org/10.5281/zenodo.1172988

DS Katz, NP Chue Hong, T Clark, A Muench, S Stall, D Bouquin, M Cannon, S Edmunds, T Faez, P Feeney, M Fenner, M Friedman, G Grenier, M Harrison, J Heber, A Leary, C MacCallum, H Murray, E Pastrana, K Perry, D Schuster, M Stockhause, and J Yeston. 2021. Recognizing the value of software: a software citation guide [version 2; peer review: 2 approved]. *F1000Research* 9, 1257 (2021). https://doi.org/10.12688/f1000research.26932.2

Diane Kelly. 2013. Industrial Scientific Software: A Set of Interviews on Software Development. In *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research* (Ontario, Canada) *(CASCON '13)*. IBM Corp., Riverton, NJ, USA, 299–310. http://dl.acm.org/citation.cfm?id=2555523.2555555

Diane Kelly. 2015. Scientific software development viewed as knowledge acquisition: Towards understanding the development of risk-averse scientific software. *Journal of Systems and Software* 109 (2015), 50–61. https://doi.org/10.1016/j.jss.2015.07.027

Diane Kelly and Terry Shepard. 2000. Task-directed software inspection technique: an experiment and case study. In *CASCON '00: Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research* (Mississauga, Ontario, Canada). IBM Press, 6. http://portal.acm.org/citation.cfm?id=782040#

Diane F. Kelly and Rebecca Sanders. 2008. Assessing the Quality of Scientific Software. In *Proceedings of the First International Workshop on Software Engineering for Computational Science and Engineering (SECSE 2008)*. In conjunction with the 30th International Conference on Software Engineering (ICSE), Leipzig, Germany. http://www.cse.msstate.edu/~SECSE08/schedule.htm

Diane F. Kelly, W. Spencer Smith, and Nicholas Meng. 2011. Software Engineering for Scientists. *Computing in Science & Engineering* 13, 5 (Oct. 2011), 7–11.

Philippe Kruchten, Robert L Nord, and Ipek Ozkaya. 2012. Technical debt: From metaphor to theory and practice. *IEEE Software* 29, 6 (2012), 18–21.

Marc J. Kuchner. 2012. *Marketing for Scientists: How to Shine in Tough Times*. Island Press, Washington, D.C.

Jonas Latt, Orestis Malaspinas, Dimitrios Kontaxakis, Andrea Parmigiani, Daniel Lagrava, Federico Brogi, Mohamed Ben Belgacem, Yann Thorimbert, Sébastien Leclaire, Sha Li, et al. 2021. Palabos: parallel lattice Boltzmann solver. *Computers & Mathematics with Applications* 81 (2021), 334–350.

Jörg Lenhard, Simon Harrer, and Guido Wirtz. 2013. Measuring the installability of service orchestrations using the square method. In *2013 IEEE 6th International Conference on Service-Oriented Computing and Applications*. IEEE, 118–125.

T.C. Lethbridge, J. Singer, and A. Forward. 2003. How software engineers use documentation: the state of the practice. *IEEE Software* 20, 6 (2003), 35–39. https://doi.org/10.1109/MS.2003.1241364

Maximilien Levesque, Magali Duvail, Ignacio Pagonabarraga, Daan Frenkel, and Benjamin Rotenberg. 2013. Accounting for adsorption and desorption in lattice Boltzmann simulations. *Physical Review E* 88, 1 (2013), 013308.

A. Logg, K.-A. Mardal, and G. N. Wells (Eds.). 2012. *Automated Solution of Differential Equations by the Finite Element Method*. Lecture Notes in Computational Science and Engineering, Vol. 84. Springer. https://doi.org/10.1007/978-3-642-23099-8

D. V. Luciv, D. V. Koznov, G. A. Chernishev, A. N. Terekhov, K. Yu. Romanovsky, and D. A. Grigoriev. 2018. Detecting Near Duplicates in Software Documentation. *Programming and Computer Software* 44, 5 (01 Sep 2018), 335–343. https://doi.org/10.1134/S0361768818050079

Bazargul Matkerim, Darhan Akhmed-Zaki, and Manuel Barata. 2013. Development High Performance Scientific Computing Application Using Model-Driven Architecture. *Applied Mathematical Sciences* 7, 100 (2013), 4961–4974.

Marco D Mazzeo and Peter V Coveney. 2008. HemeLB: A high performance parallel lattice-Boltzmann code for large scale fluid flow in complex geometries. *Computer Physics Communications* 178, 12 (2008), 894–914.

J. McCall, P. Richards, and G. Walters. 1977. *Factors in Software Quality*. NTIS AD-A049-014, 015, 055.

Mike McQuaid. 2018. The Open Source Contributor Funnel (or: Why People Don't Contribute To Your Open Source Project). https://mikemcquaid.com/2018/08/14/the-open-source-contributor-funnel-why-people-dont-contribute-to-your-open-source-project/.

Peter Michalski. 2021. *State of The Practice for Lattice Boltzmann Method Software*. Master's thesis. McMaster University, Hamilton, Ontario, Canada.

JD Musa, Anthony Iannino, and Kazuhira Okumoto. 1987. Software reliability: prediction and application.

Udit Nangia and Daniel S. Katz. 2017. Track 1 Paper: Surveying the U.S. National Postdoctoral Association Regarding Software Use and Training in Research. Zenodo, 1–6. https://doi.org/10.5281/zenodo.814220 This paper was submitted to WSSSPE5.1 - http://wssspe.researchcomputing.org.uk/wssspe5-1/ The final accepted version is https://doi.org/10.6084/m9.figshare.5328442.

Luke Nguyen-Hoan, Shayne Flint, and Ramesh Sankaranarayana. 2010. A Survey of Scientific Software Development. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement* (Bolzano-Bozen, Italy) *(ESEM '10)*. ACM, New York, NY, USA, Article 12, 10 pages. https://doi.org/10.1145/1852786.1852802

Jakob Nielsen. 2012. Usability 101: Introduction to Usability. https://www.nngroup.com/articles/usability-101-introduction-to-usability/

Ileana Ober, Marc Palyart, Jean-Michel Bruel, and David Lugato. 2018. On the use of models for high-performance scientific comput-

ing applications: an experience report. *Software & Systems Modeling* 17, 1 (01 Feb 2018), 319–342. https://doi.org/10.1007/s10270-016-0518-0

Pablo Orviz, Álvaro López García, Doina Cristina Duma, Giacinto Donvito, Mario David, and Jorge Gomes. 2017. A set of common software quality assurance baseline criteria for research projects. https://doi.org/10.20350/digitalCSIC/12543

David L. Parnas. 1972. On the Criteria To Be Used in Decomposing Systems into Modules. *Comm. ACM* 15, 2 (Dec. 1972), 1053–1058.

David Lorge Parnas. 1976. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering* 1 (1976), 1–9.

Stephen R Piccolo and Michael B Frampton. 2016. Tools and techniques for computational reproducibility. *GigaScience* 5, 1 (07 2016). https://doi.org/10.1186/s13742-016-0135-4 arXiv:https://academic.oup.com/gigascience/article-pdf/5/1/s13742-016-0135-4/25513324/13742_2016_article_135.pdf s13742-016-0135-4.

Gustavo Pinto, Igor Steinmacher, and Marco Aurélio Gerosa. 2016. More Common Than You Think: An In-depth Study of Casual Contributors. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 112–123. https://doi.org/10.1109/SANER.2016.68

Gustavo Pinto, Igor Wiese, and Luis Felipe Dias. 2018. How Do Scientists Develop and Use Scientific Software? An External Replication. In *Proceedings of 25th IEEE International Conference on Software Analysis, Evolution and Reengineering*. 582–591. https://doi.org/10.1109/SANER.2018.8330263

Gede Artha Azriadi Prana, Christoph Treude, Ferdian Thung, Thushari Atapattu, and David Lo. 2018. Categorizing the Content of GitHub README Files. arXiv:1802.06997 [cs.SE]

Brian Proffitt. 2017. How to Attract New Contributors. https://www.redhat.com/en/blog/how-attract-new-contributors.

Markus Püschel, Bryan Singer, Manuela Veloso, and José M. F. Moura. 2001. Fast Automatic Generation of DSP Algorithms. In *International Conference on Computational Science (ICCS) (Lecture Notes In Computer Science, Vol. 2073)*. Springer, 97–106.

Patrick J. Roache. 1998. *Verification and Validation in Computational Science and Engineering*. Hermosa Publishers, Albuquerque, New Mexico.

Suzanne Robertson and James Robertson. 1999. *Mastering the Requirements Process*. ACM Press/Addison-Wesley Publishing Co, New York, NY, USA, Chapter Volere Requirements Specification Template, 353–391.

J. Rokicki and L. Laniewski-Wollk. 2016. Adjoint lattice Boltzmann for topology optimization on multi-GPU architecture. *Computers & Mathematics with Applications* 71, 3 (2016), 833–848.

Mariusz Rutkowski, Wojciech Gryglas, Jacek Szumbarski, Christopher Leonardi, and Łukasz Łaniewski-Wołłk. 2020. Open-loop optimal control of a flapping wing using an adjoint Lattice Boltzmann method. *Computers & Mathematics with Applications* 79, 12 (2020), 3547–3569.

Reza Sadeghi, Seyedvahid Khodaei, Javier Ganame, and Zahra Keshavarz-Motamed. 2020. Towards non-invasive computational-mechanics and imaging-based diagnostic framework for personalized cardiology for coarctation. *Scientific Reports* 10, 1 (2020), 9048. https://doi.org/10.1038/s41598-020-65576-y

Rebecca Sanders and Diane Kelly. 2008. Dealing with Risk in Scientific Software Development. *IEEE Software* 4 (July/August 2008), 21–28.

Tobias Schlauch, Michael Meinel, and Carina Haupt. 2018. DLR Software Engineering Guidelines. https://doi.org/10.5281/zenodo.1344612

Sebastian Schmieschek, Lev Shamardin, Stefan Frijters, Timm Krüger, Ulf D Schiller, Jens Harting, and Peter V Coveney. 2017. LB3D: A parallel implementation of the Lattice-Boltzmann method for simulation of interacting amphiphilic fluids. *Computer Physics Communications* 217 (2017), 149–161.

Michael A Seaton, Richard L Anderson, Sebastian Metz, and William Smith. 2013. DL_MESO: highly scalable mesoscale simulations. *Molecular Simulation* 39, 10 (2013), 796–821.

Judith Segal. 2005. When Software Engineers Met Research Scientists: A Case Study. *Empirical Software Engineering* 10, 4 (Oct. 2005), 517–536. https://doi.org/10.1007/s10664-005-3865-y

Judith Segal and Chris Morris. 2008. Developing Scientific Software. *IEEE Software* 25, 4 (July/August 2008), 18–20.

Arfon Smith. 2022a. Enhanced support for citations on GitHub. https://github.blog/2021-08-19-enhanced-support-citations-github/.

Arfon M Smith, Daniel S Katz, Kyle E Niemeyer, and FORCE11 Software Citation Working Group. 2016a. Software Citation Principles. *PeerJ Preprints* 4 (Aug. 2016), e2169v4. https://doi.org/10.7287/peerj.preprints.2169v4

Arfon M. Smith, Kyle E. Niemeyer, Daniel S. Katz, Lorena A. Barba, George Githinji, Melissa Gymrek, Kathryn D. Huff, Christopher R. Madan, Abigail Cabunoc Mayes, Kevin M. Moerman, Pjotr Prins, Karthik Ram, Ariel Rokem, Tracy K. Teal, Roman Valls Guimera, and Jacob T. Vanderplas. 2018c. Journal of Open Source Software (JOSS): design and first-year review. *PeerJ Computer Science* 4 (12 Feb. 2018), e147+. https://doi.org/10.7717/peerj-cs.147

Barry Smith, Roscoe Bartlett, and xSDK Developers. 2018a. xSDK Community Package Policies. https://doi.org/10.6084/m9.figshare.4495136.v6

Spencer Smith. 2022b. Software Quality Grades for Lattice Boltzmann Solvers. https://data.mendeley.com/datasets/5dym63wn6z/1. https://doi.org/10.17632/5dym63wn6z.1

Spencer Smith, Yue Sun, and Jacques Carette. 2015. State of the practice for developing oceanographic software. *McMaster University, Department of Computing and Software* (2015).

W. Spencer Smith. 2016. A Rational Document Driven Design Process for Scientific Computing Software. In *Software Engineering for Science*, Jeffrey C. Carver, Neil Chue Hong, and George Thiruvathukal (Eds.). Chapman and Hall/CRC, Boca Raton, FL, Chapter Examples of the Application of Traditional Software Engineering Practices to Science, 33–63.

W. Spencer Smith. 2018. Beyond Software Carpentry. In *2018 International Workshop on Software Engineering for Science (held in conjunction with ICSE'18)*. 1–8.

W Spencer Smith, Jacques Carette, and John McCutchan. 2008. Commonality analysis of families of physical models for use in scientific computing. In *Proceedings of the First International Workshop on Software Engineering for Computational Science and Engineering (SECSE08)*.

W. Spencer Smith, Jacques Carette, Peter Michalski, Ao Dong, and Oluwaseun Owojaiye. 2021. Methodology for Assessing the State of the Practice for Domain X. https://arxiv.org/abs/2110.11575.

W. Spencer Smith and Chien-Hsien Chen. 2004. Commonality and Requirements Analysis for Mesh Generating Software. In *Proceedings of the*

Sixteenth International Conference on Software Engineering and Knowledge Engineering (SEKE 2004), F. Maurer and G. Ruhe (Eds.). Banff, Alberta, 384–387.

W. Spencer Smith and Nirmitha Koothoor. 2016. A Document-Driven Method for Certifying Scientific Computing Software for Use in Nuclear Safety Analysis. *Nuclear Engineering and Technology* 48, 2 (April 2016), 404–418. https://doi.org/10.1016/j.net.2015.11.008

W. Spencer Smith and Lei Lai. 2005. A New Requirements Template for Scientific Computing. In *Proceedings of the First International Workshop on Situational Requirements Engineering Processes – Methods, Techniques and Tools to Support Situation-Specific Requirements Engineering Processes, SREP'05*, J. Ralyté, P. Ågerfalk, and N. Kraiem (Eds.). In conjunction with 13th IEEE International Requirements Engineering Conference, Paris, France, 107–121.

W. Spencer Smith, Lei Lai, and Ridha Khedri. 2007. Requirements Analysis for Engineering Computation: A Systematic Approach for Improving Software Reliability. *Reliable Computing, Special Issue on Reliable Engineering Computation* 13, 1 (Feb. 2007), 83–107. https://doi.org/10.1007/s11155-006-9020-7

W. Spencer Smith, Adam Lazzarato, and Jacques Carette. 2016b. State of Practice for Mesh Generation Software. *Advances in Engineering Software* 100 (Oct. 2016), 53–71.

W. Spencer Smith, Adam Lazzarato, and Jacques Carette. 2018b. State of the Practice for GIS Software. https://arxiv.org/abs/1802.03422.

W. Spencer Smith, John McCutchan, and Jacques Carette. 2017. *Commonality Analysis for a Family of Material Models*. Technical Report CAS-17-01-SS. McMaster University, Department of Computing and Software.

W. Spencer Smith, Yue Sun, and Jacques Carette. 2018d. Statistical Software for Psychology: Comparing Development Practices Between CRAN and Other Communities. https://arxiv.org/abs/1802.07362. 33 pp.

W. Spencer Smith, Zheng Zeng, and Jacques Carette. 2018e. Seismology Software: State of the Practice. *Journal of Seismology* 22, 3 (May 2018), 755–788.

SourceLevel. 2022. What is a linter and why your team should use it? https://sourcelevel.io/blog/what-is-a-linter-and-why-your-team-should-use-it.

Graeme Stewart et al. 2017. A Roadmap for HEP Software and Computing R&D for the 2020s. *arXiv* (2017). arXiv:1712.06982 [physics.comp-ph]

Tim Storer. 2017. Bridging the Chasm: A Survey of Software Engineering Practice in Scientific Programming. *ACM Comput. Surv.* 50, 4, Article 47 (Aug. 2017), 32 pages. https://doi.org/10.1145/3084225

Keenan Szulik. 2017. Don't judge a project by its GitHub stars alone. https://blog.tidelift.com/dont-judge-a-project-by-its-github-stars-alone.

Daniel Szymczak, W. Spencer Smith, and Jacques Carette. 2016. Position Paper: A Knowledge-Based Approach to Scientific Software Development. In *Proceedings of SE4Science'16 in conjunction with the International Conference on Software Engineering (ICSE)*. In conjunction with ICSE 2016, Austin, Texas, United States. 4 pp.

Carsten Thiel. 2020. EURISE Network Technical Reference. https://technical-reference.readthedocs.io/en/latest/.

USGS. 2019. USGS Software Plannning Checklist. https://www.usgs.gov/media/files/usgs-software-planning-checklist.

Maarten van Gompel, Jauco Noordzij, Reinier de Valk, and Andrea Scharnhorst. 2016. Guidelines for Software Quality, CLARIAH Task Force 54.100. https://github.com/CLARIAH/software-quality-guidelines/blob/master/softwareguidelines.pdf.

Hans van Vliet. 2000. *Software Engineering (2nd ed.): Principles and Practice*. John Wiley & Sons, Inc., New York, NY, USA.

Todd. L. Veldhuizen. 1998. Arrays in Blitz++. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98), Lecture Notes in Computer Science*. Springer-Verlag.

Florian Weik, Rudolf Weeber, Kai Szuttor, Konrad Breitsprecher, Joost de Graaf, Michael Kuron, Jonas Landsgesell, Henri Menke, David Sean, and Christian Holm. 2019. ESPResSo 4.0–an extensible software package for simulating soft matter systems. *The European Physical Journal Special Topics* 227, 14 (2019), 1789–1816.

David M Weiss. 1997. Defining families: The commonality analysis. *submitted to IEEE Transactions on Software Engineering* (1997). http://www.research.avayalabs.com/user/weiss/Publications.html

David M Weiss. 1998. Commonality analysis: A systematic process for defining families. In *International Workshop on Architectural Reasoning for Embedded Systems*. Springer, 214–222. citeseer.ist.psu.edu/13585.html

R. C. Whaley, A. Petitet, and J. J. Dongarra. 2001. Automated empirical optimization of software and the ATLAS project. *Parallel Comput.* 27, 1–2 (2001), 3–35.

Wiegers. 2003. *Software Requirements, 2e*. Microsoft Press.

I. S. Wiese, I. Polato, and G. Pinto. 2019. Naming the Pain in Developing Scientific Software. *IEEE Software* (2019), 1–1. https://doi.org/10.1109/MS.2019.2899838

Wikipedia. 2022. Lint (software). https://en.wikipedia.org/wiki/Lint_(software).

Greg Wilson. 2016. Software Carpentry: lessons learned [version 2; referees: 3 approved]. *F1000Research* 3, 62 (2016), 1–12.

Greg Wilson, D. A. Aruliah, C. Titus Brown, Neil P. Chue Hong, Matt Davis, Richard T. Guy, Steven H. D. Haddock, Kathryn D. Huff, Ian M. Mitchell, Mark D. Plumbley, Ben Waugh, Ethan P. White, and Paul Wilson. 2014. Best Practices for Scientific Computing. *PLoS Biol* 12, 1 (Jan. 2014), e1001745. https://doi.org/10.1371/journal.pbio.1001745

Greg Wilson, Jennifer Bryan, Karen Cranston, Justin Kitzes, Lex Nederbragt, and Tracy K. Teal. 2016. Good Enough Practices in Scientific Computing. *CoRR* abs/1609.00037 (2016). http://arxiv.org/abs/1609.00037

Greg Wilson and Andrew Lumsdaine. 2006. Software Carpentry: Getting Scientists to Write Better Code by Making Them More Productive. *Computing in Science Engineering* 8, 6 (Nov. 2006), 66–69. https://doi.org/10.1109/MCSE.2006.122

Gregory V. Wilson. 2006. Where's the Real Bottleneck in Scientific Computing? Scientists would do well to pick some tools widely used in the software industry. *American Scientist* 94, 1 (2006). http://www.americanscientist.org/issues/pub/wheres-the-real-bottleneck-in-scientific-computing

Yo Yehudi. 2021. Open Source for Researchers - Lightning talk. https://doi.org/10.5281/zenodo.5655023

Moshe Zadka. 2018. How to open source your Python library. https://opensource.com/article/18/12/tips-open-sourcing-python-libraries.

Duo Zhang, Qiong Cai, and Sai Gu. 2018. Three-dimensional lattice-Boltzmann model for liquid water transport and oxygen diffusion in cathode

of polymer electrolyte membrane fuel cell with electrochemical reaction. *Electrochimica Acta* 262 (2018), 282–296. https://doi.org/10.1016/j.electacta.2017.12.189