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 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 103 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 top three packages were ESPResSo, Ludwig, and Palabos (all in the top 10 by star count). 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. Based on our assessment, we make the following recommendations for current and future LBM projects: provide a detailed user manual and tutorials, explicitly state the limits of the software, use user-friendly programming languages, consider a peer review process, communicate development and contribution information, and use continuous integration and project management tools. [Change the abstract to discuss how LBM compares to common practices and best practices. Highlight some future practices. [Follow the general research software trend of increasing use of version control systems and project management on GitHub or similar. —SS] —SS]

Keywords: Lattice Boltzmann Method (LBM), scientific computing, 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 behavior 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), modeling 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 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

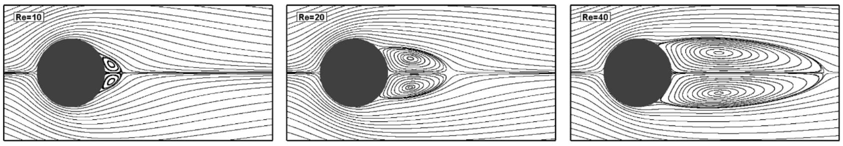

Figure 1: Streamlines of flow past a stationary circular cylinder at Reynolds number = 10, 20, and 40 (Chen et al., 2021)

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 watching for areas for potential future improvement.

To focus our efforts, we devised 11 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, and 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 by borrowing successful ideas from other domains. For LBM developers interested in future cutting edge practices, ideas that are proposed, but not yet in practice, should also be identified. 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:

- RQ9.a what software development concerns exist; and, (Section 9)
- RQ9.b what specific practices are taken to address these concerns? (Section 10)

RQ10: What, if any, existing research software best practice, not currently used by LBM software, could address the concerns identified in RQ9). (Section 10)

RQ11: What practices could be employed by future LBM developers to address the pain points found in RQ7 and the concerns identified in RQ9.a? (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., 2018a), Mesh Generators (Smith et al., 2016b), Oceanographic Software (Smith et al., 2015), Seismology software (Smith et al., 2018c), statistical software for psychology (Smith et al., 2018b) 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 10). 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 scientific software (Smith et al., 2021). This methodology can be instantiated for a specific domain of scientific 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

We adopt software quality definitions from various researchers and subject matter experts. Some of the definitions are from Smith et al. ([n.d.]). The following are the software quality definitions used in this exercise, along with comments regarding their quantitative and qualitative measurement. [Update when the Quality Definition of Qualities document is completed. —SS]

2.1.1. *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). A good measure of installability correlates with scenarios when low or moderate effort is required to gather and prepare software for its general use on a system for which it was designed. 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. The ability to reasonably validate the installation procedure and the ease of installation also have a positive effect on the measure of installability.

2.1.2. *Correctness*

A software program is correct if it behaves according to its stated specifications (Ghezzi et al., 2003). This requires that the specification is available. Scientific software is unlikely to have a formal specification, since this is not common practice in the field. As a consequence, the correctness of software often cannot be measured directly. Despite an absent specification, the correctness can indirectly be assessed by looking at the output and applying domain knowledge. For our measurement of correctness we have assumed that the following factors suggest correctness was considered: availability of a requirements specification, reference to domain theory, and explicit use of tools and/or techniques for building confidence of correctness, such as documentation generators and software analysis tools.

2.1.3. *Verifiability*

Verifiability is measured by the extent to which a set of tests can be written and executed to demonstrate that the delivered system meets the specification (Sommerville, 2011). 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 documentation, and with evidence of continuous integration during the development process. In our process we measure correctness and verifiability together.

2.1.4. *Reliability*

Reliability is measured by the probability of failure-free operation of a computer program in a specified environment for a specified time; that is, reliability can be measured by the average time interval between two failures, also known as the mean time to failure (MTTF) (Ghezzi et al., 2003) (Musa et al., 1987). Reliability is thus positively correlated with the absence of errors during installation and use. Recoverability from errors also improves reliability.

2.1.5. *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., 1991). 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. A reasonable reaction includes an appropriate error message and the ability to recover the system.

2.1.6. *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 that alludes to a consideration of performance, such as parallelization tools.

2.1.7. *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 a high usability correlates with the presence of documentation, including tutorials, manuals, and defined user characteristics, and user support. Preferably the user support model has avenues to contact developers and report issues.

2.1.8. 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 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.

2.1.9. 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 had been taken.

2.1.10. 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 measure of reusability results from a large number of easily reusable components. Increased software modularization, defined as the presence of smaller components with well defined interfaces, is important. For this state of the practice assessment, a good measure of reusability correlates with an increased number of code files, and the availability of API documentation.

2.1.11. 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 clarity of comments.

2.1.12. 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 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.

2.1.13. 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., 1991). 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 package version release notes.

2.1.14. Reproducibility

Software achieves reproducibility if another developer can take the requirements documentation and re-obtain the same software artifacts (Benureau and Rougier, 2017). This includes the output of the software, where the scientific results are compared between software implementations, or between software implementations and manually calculated results. 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.

2.1.15. *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 think that the current documentation can clearly convey all necessary knowledge to the users, and how they achieved this 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. Survey the developers. (Section 2.7)
7. Use AHP to rank the software packages. (Section 2.6)
8. Analyze the results and answer the research questions.

The above steps depend on interaction with a domain expert partner, as discussed in Section 2.8.

2.3. *Identify Candidate Software*

Part of how we answer RQ1. The candidate software was found through search engine queries targeting authoritative lists of software. We found LBM software listed on the websites GitHub and swMATH, as well as through articles found in scholarly journals and databases. The Domain Expert (Section 2.8) was also engaged in selecting the candidate software. [Packages that fit our criteria but were found after data collection and analysis was conducted, such as Musubi (Hasert et al., 2014a), will need to be added to future SOP assessments. —SS]

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 development phase.

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

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., 2014b)
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

2.4. *Filter the Software List*

With previous section, how we answer RQ1. Answer to RQ1 provided in this section. To reduce the number of members in the candidate software list to a manageable size, the following filters were applied. The 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 month. Packages were categorized as 'dead' if the last update of this information was more than 18 month ago.

While the initial list had 46 packages, filtering by scope, usage, and age decreased the size of the list to 24 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 24 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. The final list of software packages that were analyzed in this project can be found in the following two tables. Table 1 lists packages that fell into the 'alive' category as of mid 2020, and Table 2 lists packages that were 'dead' at that time.

There is considerable variation among these software packages, including their intended purpose, size, user interfaces, and software languages used. 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
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*

Part of how we address RQ2 and RQ4. Once have RQ2 can combine with estimate of popularity to answer RQ3. The measurement template is found in Smith et al. (2021). This template is used to track measurements and quality scores for all of the software packages in the domain. For each software package, 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 for measuring a given domain is feasible for a team as small as one, since we aim for a cap of 160 person hours for the measurement phase Smith et al. (2021). An excerpt of the spreadsheet is shown in Figure 2. A column is added to this template for each of the 24 software package measured.

	A	B	C	D
1	**Metrics & Description**	**Possible Measurement Values**		
2	**Summary Information**	*** is used to indicate that a response of this type should be accompanied by explanatory text**	**{software package 1}**	**{software package 2}**
3	Software name?	(string)		
4	URL?	(URL)		
5	Affiliation (institution(s))	(string or {N/A})		
6	Software purpose	(string)		
7	Number of developers (all developers that have contributed at least one commit to the project) (use repo commit logs)	(number)		
8	How is the project funded?	(unfunded, unclear, funded*) where * requires a string to say the source of funding		
9	Initial release date?	(date)		
10	Last commit date?	(date)		
11	Status? (alive is defined as presence of commits in the last 18 months)	({alive, dead, unclear})		
12	License?	({GNU GPL, BSD, MIT, terms of use, trial, none, unclear, other*}) * given via a string		
13	Platforms?	(set of {Windows, Linux, OS X, Android, other*}) * given via string		
14	Software Category? The concept category includes software that does not have an officially released version. Public software has a released version in the public domain. Private software has a released version available to authorized users only.	({concept, public, private})		
15	Development model?	({open source, freeware, commercial, unclear})		
16	Publications about the software? Refers to publications that have used or mentioned the software	(number or {unknown})		

Sheet1

Figure 2: Excerpt of the Top Section of the Measurement Template (Summary Information) [Should update this figure to something easier to read —SS]

The full template consists of 108 questions categorized under 9 qualities: 1. installability; 2. correctness and verifiability; 3. surface reliability; 4. surface robustness; 5. surface usability; 6. maintainability; 7. reusability; 8. surface understandability; and, 9. 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 of the quality. For instance, we are not currently doing any experiments to measure usability. Instead, we are looking for an indication that usability was considered by the developers. We do this by looking for cues in the documentation, like a getting started manual, a user manual and documentation of expected user characteristics.

Most of the data was gathered by manually investigating the software, its source code, and its artifacts, while some was gathered using the automatic repository measurement tools. Tools were use to find data 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 environments 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. When filling in the measurement template spreadsheet, the details for each VM should be noted, including hypervisor and operating system version.

2.6. Analytical Hierarchy Process

Part of how we address RQ2. The Analytical Hierarchy Process (AHP) is a decision-making technique that is used to compare multiple options by multiple criteria. In our work 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 for conducting this process. The tool includes a sensitivity analysis that was used 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

Part of how we address RQ5, RQ6, RQ7 (which enables RQ8, and RQ9 (which enables RQ10). We designed a list of 20 questions to guide our interviews, which can be found in Smith et al. (2021). Some questions are about the background of the software, the development teams, the interviewees, and how they organize the projects. We also ask about the developer's understandings 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 and current situations of 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 usually bring up some exciting ideas that we did not expect, and it is worth expanding on these topics.

We requested interviews from all 24 packages, except Munubi, since Munubi was added after our ethics approved study window had expired. We were able to recruit 4 developers to participate in our study. To send out interview requests we found we found contacts on the projects' website, or code repository, or publications, or the biographic pages of the teams' institutions. We send at most two interview request emails to a contact for each software package. Meeting will typically be held using on-line meeting software, like Zoom or Teams. This facilitates recording and automatic transcription of the meetings.

The interviews followed the standard ethics guideline of asking for consent before interviewing, recording, and including participant details in the report. The interview process presented here was approved by the McMaster University Research Ethics Board under the application number MREB#: 5219.

2.8. *Interaction With Domain Expert*

Part of how we address RQ1 and RQ2. [Should ask Zahra about RQ3, see if he has anything to add about RQ7. —SS] 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

Answers RQ2. Before discussing quality differences between the packages and the results of the AHP ranking, we will contrast the different packages based on features, as shown in Table 3. [Add some discussion of some example features. Add Non-Newtonian and Fluid Structure interaction. Listed in alphabetical order. —SS] The sections below summarize how the software packages compare based on qualities, as measured by the measurement template. The results of the AHP ranking are also discussed.

3.1. *Installability*

All of the 24 software packages that were tested have installation instructions. As noted previously, many of the 22 software packages that were part of the original long list of 46 packages were removed due to not including documentation or installation instructions. Of the 24 software packages that were tested, most had 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. Maintainability and correctness of these instructions could be improved if all the instructions were in one location.

All 24 packages on the short list can be installed on some Unix-like systems. 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 this operating system 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,

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)))

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 definitively 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 software dependency versions. 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.

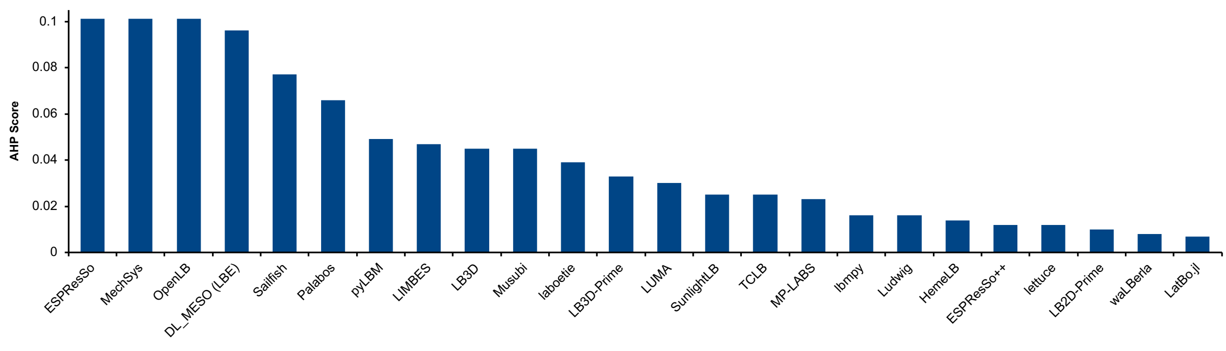

Figure 3: AHP Installability Score

Figure 3 shows the installability ranking of the software packages using AHP. Software packages with a higher score (ESPResSo, MechSys, OpenLB, DL_MESO, Palabos) tend to have one set of linear installation instructions that are 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. Furthermore, the top seven ranked packages are noted as being alive.

Many software packages would benefit from a rewrite or reorganization of installation instructions. A single location for installation instructions would improve their maintainability and correctness. Listing compatible operating system and dependency versions would decrease installation time and errors, as would adding instructions on installing dependencies. Installation process errors should prompt the system to display detailed messages. Once a software package is installed, either an automatic validation needs to be performed or the user needs to be able to perform a manual validation using test examples that include expected output. Finally, uninstallation instructions should be included in the documentation.

3.2. Surface Correctness and Verifiability

Seventeen of the software packages include a requirements specification artifact or explicitly reference domain theory, often only the latter. Software packages that distribute requirements specification information, such as DL_MESO (LBE), generally keep it brief and include it within other documentation. This artifact is often found within a user manual, on a web-page, or is mentioned in related publications. In the latter case the user may need to spend significant time to find this information. [More on requirements as a pain point is given in Weiss, and other papers cited by Carver (2021). Isn't generally requirements in the SE sense of following a template or similar. —SS]

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.

Unit tests are only explicitly available for two of the software packages, Ludwig and Musubi. Code modularization of most packages allow 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. [Forward reference that suggests more tools use CI (Section 7 Comparison of Principles, Process and Methodologies to Research Software in General). —SS]

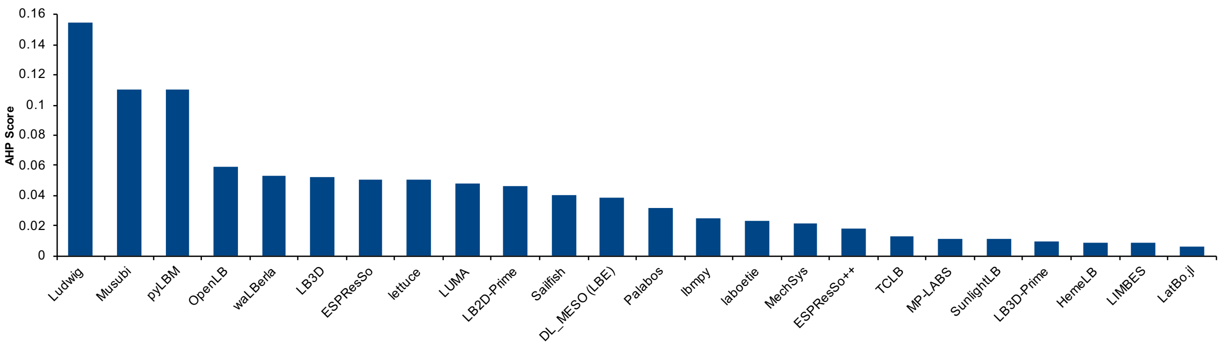

Figure 4: AHP Surface Correctness and Verifiability Score

Figure 4 shows the surface correctness and verifiability ranking of the software packages using AHP. Software packages with a higher score tend to have a visible requirements specification or references to theory documentation. They also explicitly use at least one document generation tool that builds confidence of 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 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.

The inclusion of requirements specification and theory documentation greatly benefits the correctness and verifiability of software packages. The use of document generation tools can help build confidence in correctness. The addition of easy to follow tutorials further helps users verify the software and have confidence in its correctness. Unit testing, as well as the use of continuous integration tools and techniques such as Bamboo, Jenkins, and Travis CI, help verify correctness.

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 and 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 it. 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 one that was missing a dependency was recoverable.

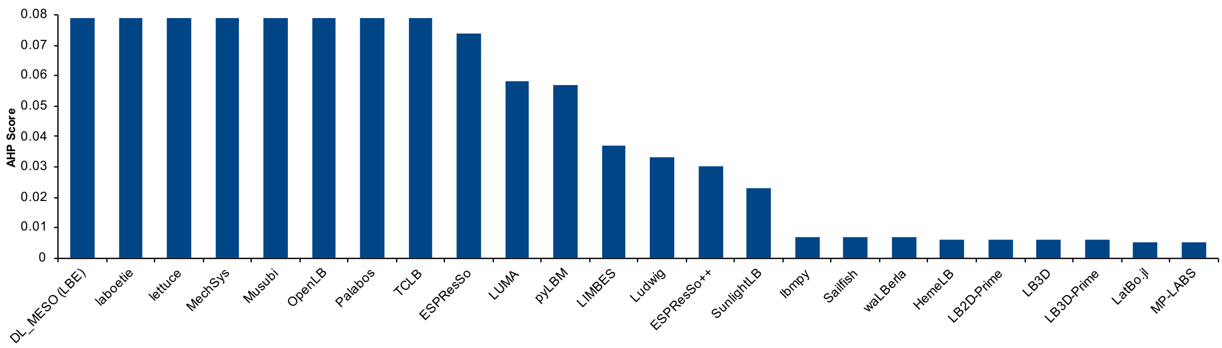

Figure 5: AHP Surface Reliability Score

Figure 5 shows the surface reliability ranking of the software packages using AHP. 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.

Overall, lower ranked software packages are lacking clear documentation, testing or tutorial examples, and descriptive error messages, and have broken dependencies. Thus, regarding surface reliability, software packages would benefit from clear up-to-date documentation that specifies all dependencies, the inclusion of testing and tutorial examples, and the assurance of descriptive error messages after faults.

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 predicated on a reasonable response from the system, including appropriate error messages and an absence of unrecoverable system failures.

Figure 6 shows the surface robustness ranking of the software packages using AHP. 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 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

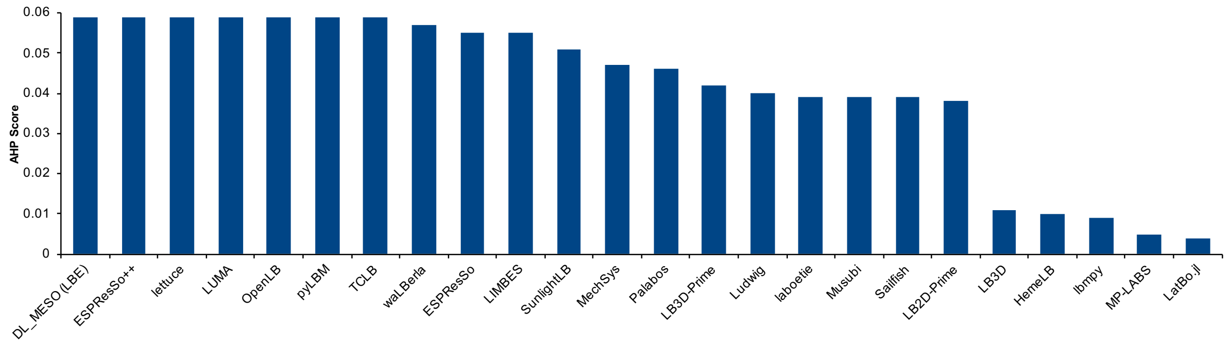

Figure 6: AHP Surface Robustness Score

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. LIMBES is noted as being classified as dead.

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 performance is not appropriate. In this project we instead 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). GPUs provide superior processing power and speed compared to CPUs, and are often used for scientific computing when a large amount of data is involved. 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. 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 on 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 shows the surface usability ranking of the software packages using AHP. 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 ranked 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 software packages were also reviewed for software release and documentation version numbers.

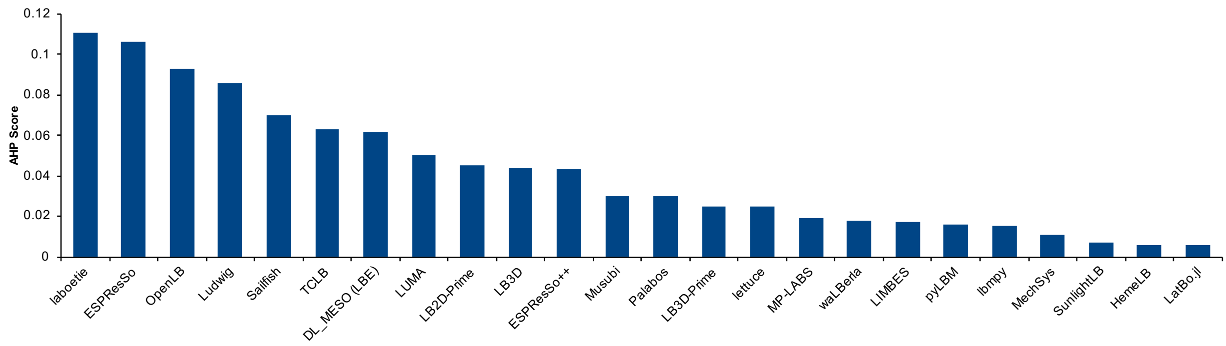

Figure 7: AHP Surface Usability Score

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 Git, seven use email, and one (SunlightLB) uses SourceForge. Most software packages that use Git have most 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 Git issue tracking) have 64% of their issues closed, while dead packages (3 use Git issue tracking) have 71% of their issues closed. This information is presented in Table 4. Furthermore, 13 packages that use Git for issue tracking use GitHub as a version control system, while two (Palabos, waLBerla) use GitLab. Of the other packages, one package (SunlightLB) uses CVS for issue tracking, 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. Packages with a higher percentage of comments were designated as 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.

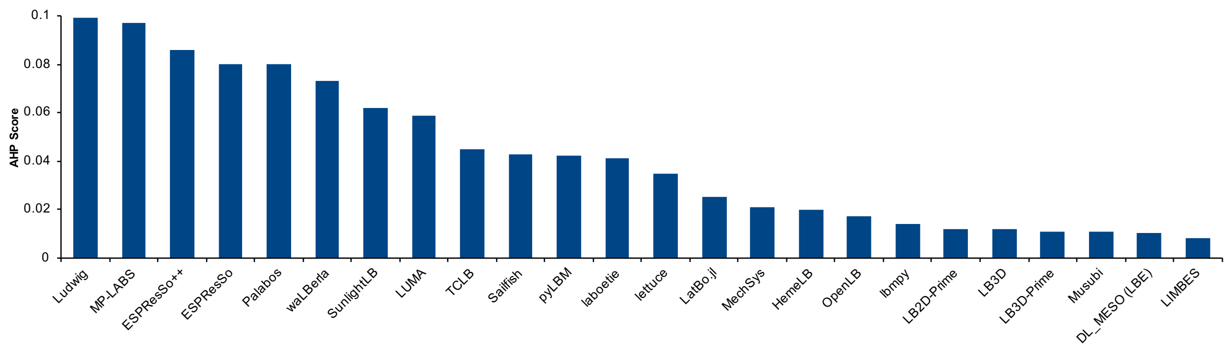

Figure 8: AHP Maintainability Score

Figure 8 shows the maintainability ranking of the software packages using AHP. Software packages with a high score provide version numbers on documents and source code releases, have an abundance of high quality artifacts,

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

and use an issue tracking tool and version control system. These packages also appear to reasonably handle issue tracking, having most of their issues closed. Their code files are well commented with more than 10 percent of the code being comments. Furthermore, four of the top 5 ranked packages are noted as being alive. MP-LABS is noted as being dead.

[Sort "Git Repository Data" table in descending order of percentage of code comments. —SS]

3.8. *Reusability*

We measured the total number of source code files for each project. A larger 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.

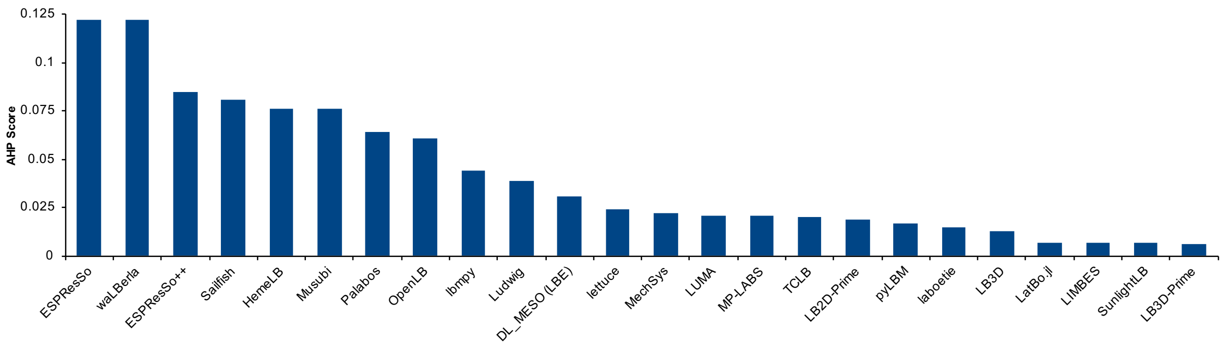

Figure 9: AHP Reusability Score

Figure 9 shows the reusability ranking of the software packages using AHP. 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 modeling. 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. Furthermore, nine of the top 10 ranked packages are noted as being alive. HemeLB is considered to be dead.

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.

There was a strong focus on modularity when designing the waLBerla framework 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).

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

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

Table 5: Module Data

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.

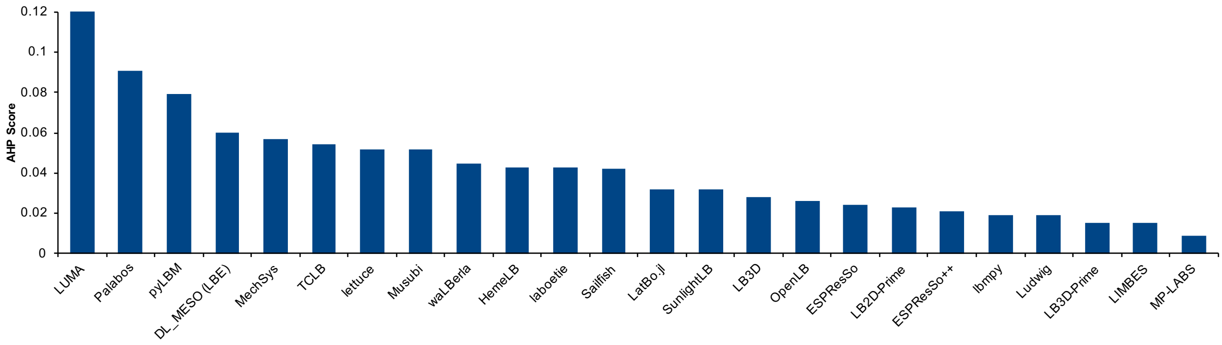

Figure 10: AHP Understandability Score

Figure 10 shows the surface understandability ranking of the software packages using AHP. 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. Furthermore, 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 standard. 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, as detailed below. 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.

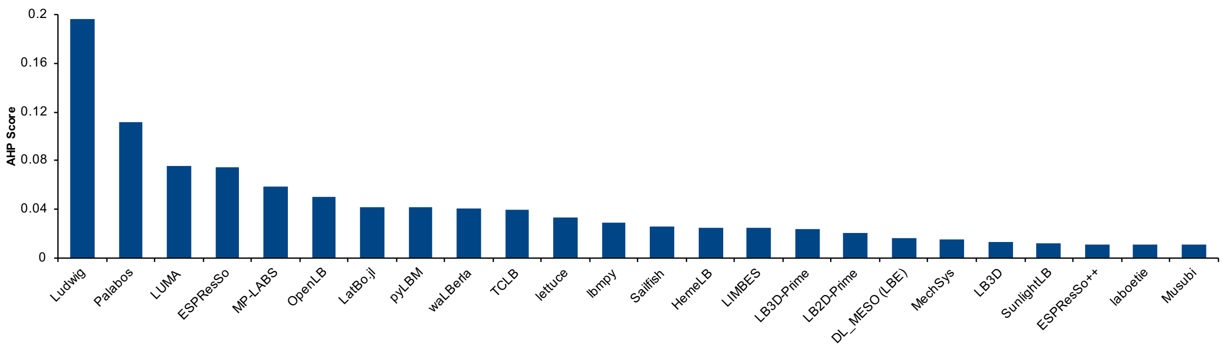

Figure 11: AHP Visibility and Transparency Score

Figure 11 shows the visibility and transparency ranking of the software packages using AHP. Software packages with a high score have an explicit development model and defined development process. They also had detailed and easy to access notes accompanying software releases. Furthermore, four of the top five ranked packages are noted as being alive. MP-LABS is noted as being dead.

3.11. Overall Quality

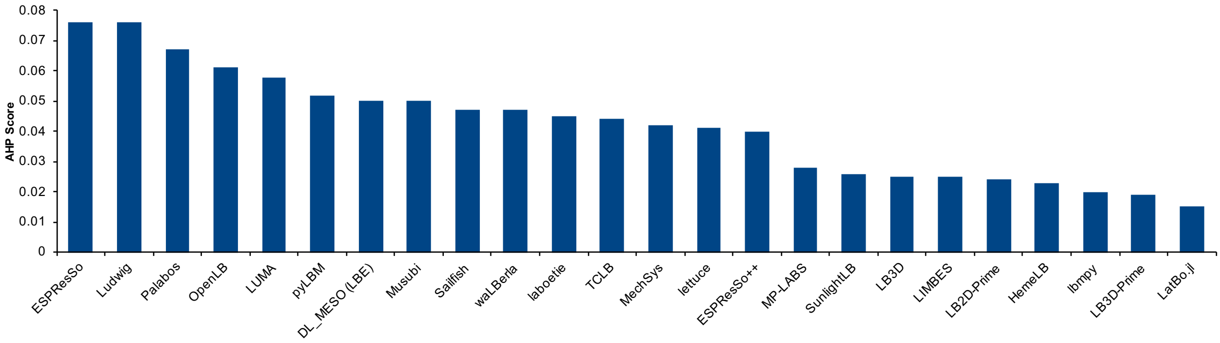

Figure 12: AHP Overall Score

Figure 12 shows the overall ranking of the software packages using AHP. 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 had achieved a relative 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

Answers RQ3. Table 6 presents our LBM software package rankings along with the repository ranking metrics of each software package. 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 slight pattern where packages that have been highly ranked by our assessment do have more stars than lower ranked packages. The best ranked package (ESPResSo) has the second most number of stars. The ninth ranked package (Sailfish) has the most number of stars. Packages designated as lower quality often do not use GitHub or GitLab, or have few stars. HemeLB only has 12 stars, but its location recently moved from a different GitHub repository. Our assessment of this package might not be accurate. The number of stars does not necessarily represent the perceptions of the community, but for lack of an alternative measure we will use it this way. Besides missing data, another threat to the validity of this comparison is the varying ages of the repositories. Older packages have been able to accumulate stars and watches for longer than newer packages. The true quality of new packages may not be reflected in their stars and watches.

The repository watches column contains even less data to compare since two of the packages (Palabos, waLBerla) use GitLab, which does not include this metric. The pattern that can be observed with this metric is very similar to that of the stars metric. The ninth ranked package (Sailfish) has the most number of watches. The best ranked package (ESPResSo) has the second most number of watches.

Table 7 compares the top 10 ranked LBM software packages with a LBM package ranking made by a domain expert. Five (Palabos, OpenLB, LUMA, Sailfish, waLBerla) of the top 10 ranked packages in this assessment are also found in the domain expert's top 10 ranking. Interestingly, the first three of these five packages are listed in the same order on both lists. Looking at the remaining packages, the top two ranked packages in this state of the practice

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

Table 6: Repository Ranking Metrics

Rank	This Assessment	Domain Expert
1	ESPResSo (N/A)	Palabos
2	Ludwig (N/A)	OpenLB
3	Palabos (1)	LUMA
4	OpenLB (2)	ASL
5	LUMA (3)	ch4-project
6	pyLBM (16)	Open FSI
7	DL_MESO (LBE) (11)	WaLBerla
8	Musubi (N/A)	LIFE
9	Sailfish (9)	Sailfish
10	WaLBerla (7)	HemeLB

Table 7: Ranking Comparison

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

Table 8: Artifacts Present in Surveyed Packages, Classified by Frequency

assessment (ESPResSo, Ludwig) are not on the domain expert's list. Perhaps the intended applications of these packages, complex fluids for Ludwig and soft matter research for ESPResSo, do not align with the research interests of the domain expert. This reasoning may also apply to eighth ranked Musubi. Moving down the list, pyLBM and DL_MESO(LBE) did not make the domain expert's top 10 list but were mentioned by the domain expert. Looking at the remainder of the domain expert's list, ASL is a general purpose tool for solving partial differential equations and may have fallen out of scope of the authoritative lists that were used to identify the initial LBM software list. The domain expert's fifth ranked package (ch4-project) was on the initial software list of this assessment, but was removed due to a lack of documentation. Open FSI is a new project that uses Palabos. It was made public the year before data was collected for this state of the practice exercise. It may not have been on authoritative lists due to its young age. Similarly, LIFE was also made public recently. Finally, the domain expert's 10th ranked package (HemeLB) was ranked 21st in this state of the practice assessment.

5. Comparison of Artifacts to Research Software in General

Answers RQ4. The software 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 8. 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.

All of the top four AHP ranked packages, ESPREsSo, Ludwig, Palabos, and OpenLB, have each of the commonly found artifacts, except only three of them have a requirements specification or theory notes. [Change to clarify that

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 9: Observed development tools, dependencies and project management tools

it is theory notes, not a requirements specification. Remove requirements spec from table and put it as rare. —SS] Palabos is the only one of these four packages that does not have an artifact from this category. [** Requirements ** —SS] [Discuss requirements here. Extra information from Peter: I interpreted it to mean what the software should do. As noted in my report, this was not common, especially finding any formal document like an SRS. I do not recall any package having an SRS. Some packages did note in their manuals or artifacts some of the points that would be found in an SRS document. For example, DL_MESO notes functional ("The following is a list of the features that DLMESO currently supports....") and non-functional (should work in both "serial and parallel running; it can be run on standalone machines, clusters and supercomputers..") requirements in its documentation (more on this below). Yes, there was often some sort of a theory resource. Often it was simply a link or reference to the theory used in the package. There was significant discrepancy in how much theory was documented or referenced. Meanwhile, requirements were rare. A few packages did note some of this information within other artifacts but it was not comprehensive, and it was not presented in a SRS document. DL_MESO has some of the information that belongs in an SRS in its user manual. It is not comprehensive or presented as it would be in an SRS, but it was probably the best example of requirements. It has a nice subsection on Functionality. Nguyen-Hoan et al. (2010) showed requirements are the least commonly produced type of documentation for scientific software in general. LBM matches this trend. —SS]

The top four AHP ranked packages 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. Despite the broken link, Palabos does have a detailed and informative website. Another one (Ludwig) of the top four packages does not appear to have publicly visible design documentation. A third package (OpenLB) from the list does not appear to use a version control system. It is possible that such a system is used by the package, since its website notes package version numbers, but the artifacts do not explicitly state the use of such a system.

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 a FAQ section or forum. One (OpenLB) has verification and validation notes, and a video guide of the software.

[Weiss mentions pain points related to design documentation. Carver (2021) also mentions design documentation in the literature review. —SS]

6. Comparison of Tool Usage to Research Software in General

Answers RQ5. Software tools are used to support the development, verification, maintenance, and evolution of software, software processes, and artifacts (Ghezzi et al., 1991). Many tools are used by LBM software packages, as summarized in Table 9. The tools are subdivided into development tools, dependencies, and project management tools.

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., 1991). Although not shown in Table 9, 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 9), 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 scientific computing libraries (e.g. SciPy). Some used libraries will be missing, if they are not explicitly stated in the artifacts we observed.

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 9 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 are noted as being used when developing the software projects. Most often email and video conferencing is used. Project management software was not explicitly mentioned, but it is possible that some of the projects use such software. Many of the projects are located on GitHub, and its 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. [**Version Control** —SS] [This contrasts with work from a little over 10 years ago that shows version control was only used in 50% of research software projects (Nguyen-Hoan et al., 2010). At that time Nguyen-Hoan et al. (2010) noted an increase from previous levels of version control software use. Anecdotally this confirms our observations. —SS] They often use GitHub or GitLab for this. One package (SunlightLB) uses CVS. Document generation tools are mentioned in the artifacts of 12 of the projects. The tools Sphinx and Doxygen are explicitly used in this capacity. [look at what the other papers say are commonly used tools and bring that information into this section. —SS]

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

Answers RQ6. Most of the software packages do not explicitly state in their artifacts 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 opting for using Python and CUDA/OpenCL to achieve this without sacrificing any computational performance. The goals of that project are explicitly listed as performance, scalability, agility and extendability, maintenance, and ease of use. The project scored well in these categories during our assessment.

Processes, like methods, are ways of doing things, especially in an orderly way; while methodologies are defined as systems of methods (Ghezzi et al., 1991). It is not explicitly indicated in the artifacts of most of the packages that development involved following any specific model, like a waterfall or agile development model. One developer (ESPResSo) noted that while no formal model is used, their development model is something similar to a combination of agile and waterfall development models. The developer teams of the LBM packages are fairly small, so it is feasible for them to be organized without the need for such models. [some repetition from the previous section - should try to figure out how to remove the repetition. —SS]

Seven of the software packages contain artifacts recording the general development process and status. Eleven of the packages explicitly convey that they would accept outside contributors, but generally the teams are centralized, often working at the same institution.

The developers that were interviewed all noted 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.

Thirteen (54%) of the 24 software packages use GitHub for managing the project, including nine (60%) of the 15 alive packages, and four of the nine (44%) dead packages. Two projects (Palabos, WaLBerla) use GitLab. This could be indicative of a transition to such software development and version control tools for SCS. Typically there are several simultaneous development branches in these projects.

Documentation was also 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. The use of document generation tools is mentioned in the artifacts of 13 software packages, and was noted during interviews with developers. Sphinx and Doxygen are the tools that were mentioned.

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, and issue tracking is commonly used to manage these changes.

Interviews with the developers of software packages also revealed a more frequent use of both unit testing and continuous integration in the development process than was found by only assessing the artifacts. The use of automatic installation processes is also common. Most often this involved a Make script.

8. Developer Pain Points

Answers RQ7 and RQ8. This subsection answers the research question: What are the pain points for developers working on research software projects? What aspects of the existing processes, methodologies and tools do they consider as potentially needing improvement? How should processes, methodologies and tools be changed to improve software development and software quality? Developers were asked to comment on obstacles in their development process, obstacles encountered by users, and potential future obstacles.

Pain points that others, but not LBM developers, mentioned in surveys include (Pinto et al., 2018): Cross-platform compatibility, interruptions while coding, scope bloat, lack of user feedback, mismatch between coding skills and subject-matter skills (for instance, "I am a mathematician, not an expert on software"), lack of formal reward system, hard to collaborate on software projects, aloneness.

8.1. 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. Team members are almost always part of the same institute or already know each other from other projects. External contributions are rare despite many of the projects accepting them. [Other projects cite the same point point - Wiese paper —SS] Aside from on-boarding new developers, time constraints could be mitigated by increasing developer efficiency, which could be addressed in several ways, including by improving the quality of documentation, or incorporating automatic code generation. [Potential ways to address the pain points might go into a separate section? Maybe propose the strategy, like code generation, and then list the pain points that are addressed? —SS]

Lack of time was also mentioned by Pinto et al. (2018) and Pinto et al. (2016).

8.2. Lack of Software Development Experience

A lack of software development experience was noted by the developer of TCLB. Many of the team members on their project are domain experts and there can be a steep learning curve before these members contribute good quality source code. This problem has been mitigated by re-writing the code to more easily enable future contributions. On a related point, improving the software engineering education or experience of developers is also an idea that was brought up by several developers. [The Wiese paper and others talk about the background of developers. The knowledge for developing research software primarily comes from peers and through self-study, not from formal training (Hannay et al., 2009). The thinking of the developer on the role of formal education matches the trend Pinto et al. (2018) observes, where in their replication of a previous study (Hannay et al., 2009), they show a growing interest in formal training (From 13% of respondants in 2009 to 22% in 2018). —SS]

Poor software documentation is also mentioned by Pinto et al. (2018) and Lethbridge et al. (2003). The survey from Lethbridge et al. (2003) is for software in general, not just research software.

8.3. Lack of Incentive and Funding

The TCLB developer noted a lack of incentive and funding in academia for developing widely used scientific software. The importance of funding for scientific software has been discussed in Gewaltig and Cannon (2012), which notes that "software tools are developed and maintained only for as long as there is explicit or implicit funding". The developer further commented that there are no journals that publish such scientific software source code. However, there are ways to get such source code cited. Work has been done to address this in (Smith et al., 2016a), which presents a set of software citation principles and discusses "how they could be used to implement software citation in the scholarly community" (Katz et al., 2019). [other papers also mention this pain point. Solutions proposed by Katz, journal of open source etc. —SS]

8.4. Lack of External Support

Another raised concern was that there are no organizations helping with the development of good quality software; but some do exist, including Better Scientific Software (BSSw), Software Sustainability Institute (Crouch et al., 2013), and Software Carpentry. Some SCS developers may not be familiar with these organizations. (Wilson and Lumsdaine, 2006)

Scientific software is often developed in-house by the very researchers that temporarily use it in their own research. Empirical studies of such "professional end-user development" of SCS is noted in (Segal, 2007). This kind of software has a defined user and purpose, and often does not meet the standards that would be required by external users. It has been categorized as a "private tool" by (Gewaltig and Cannon, 2012), which notes that despite often being made freely available, "it is not always clear that it is sufficiently mature in terms of domain coverage, validity, documentation or usability, to be useful to other researchers". It is less common for such ad hoc scientific software to become "user-ready software", which "is not only research-ready, but should have most of the attributes commonly expected of commercial software products including broadness of scope, robustness, demonstrable correctness and adequate documentation" (Gewaltig and Cannon, 2012). As software becomes user-ready, it can become commercialized and closed-source. [remove this paragraph? —SS]

[Remove this section? The external support organizations will likely come up as the mitigation for some of the other pain points. —SS]

8.5. Parallelization and Continuous Integration

Setting up parallelization was also noted as a technical pain point by one of the developers, and the introduction of continuous integration by another. Software development knowledge, and automatic code generation, could mitigate such pain points. As already noted, many of the developers are domain experts and not professional software developers. The developer of TCLB noted that eliminating equivalent statements using macros had helped improve the quality of their source code, specifically helping with reusing code to run on both the CPU and GPU. [The point about macros isn't really related to parallelization or continuous integration. —SS]

8.6. Ensuring Correctness

Difficulties with ensuring correctness were also noted by several developers. They indicated that tests are run on all new source code additions, testing both individual modules and the system to verify correctness. These tests compare the package output to known correct output using test cases. [oracle —SS] The developer of TCLB commented that the amount of testing data that is needed for some cases is a problem as free testing services do not offer capabilities to store and process such large amounts of data, and in-house testing solutions needed to be created to address this limitation. The solution for this has been to limit the size of the testing problems, and to run tests in small batches with few iterations. [The Weiss paper alludes this this as well. I think the Kelly and Sanders paper likely talks about testing problems and correctness. "while many scientists believe that software testing is important, a smaller number believe they have sufficient understanding about testing concepts" (Hannay et al., 2009). —SS]

[** BEGIN Testing and Verification ** —SS]

Nguyen-Hoan et al. (2010) says the following:

- 38 of 60 respondents performed unit tests
- 23 of 60 respondents performed integration tests
- "In our results, the four most common activities involve comparisons with known, trusted, or prior data. Verification and peer review were three of the four least commonly performed activities." [testing is not the only way to verify software. This addresses the oracle problem. —SS]

May do more testing than artifacts indicate, but that is still a problem. Artifacts should be available so that others can verify the results, verify their installation. Part of an assurance case means that a third party can verify the work.

[** END Testing and Verification ** —SS]

"The questions of what constitutes an adequate testing oracle for scientific software and what constitutes adequate test data are left open. Some of the validation testing we researched was running up against the practical limits; they were using whatever oracle they could scrape together from industry data, benchmarks, and expert judgment." (Kelly and Sanders, 2008)

[Seems that LBM not concerned with the oracle problem? —SS]

8.7. Usability

A few obstacles related to users were found. 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 the packages require both a good understanding of CFD and an understanding of the requirements for formulating problems in the individual packages, which can be a significant endeavor. These software packages are not like commercial software packages. They are generally set up to solve specific research problems, and are often primarily used by their developers. While they are modifiable to solve similar problems, these modifications are not trivial. Better documentation, with attention on traceability, and automatic code generation are suggested when designing software for change, and would help with these modifications. [Should decide if want to introduce ideas for improving within the presentation of the pain point, or separately? —SS] So far this problem of onboarding new users has been addressed by updating the documentation to better inform users of the underlying LBM theory and package requirements. Similar issues with LBM parameters were noted by another developer. Updating the user interface to better explain theoretical principles, as well as test user input for compatibility, was the implemented solution. As noted above, sometimes frequently asked questions on the underlying theory and on how to use the software are answered in the documentation. The interviewed developer of ESPResSo commented that parts of their package's source code had been refactored to Python to help address usability issues. Python was perceived as a much more usable language, and it would be easy for future users and developers to learn and understand the source code.

8.8. Technical Debt

A few potential future obstacles were noted. The developer of ESPResSo noted that their source code had been written with a specific application in mind and that due to this there was too much coupling between components of the source code. This results in technical debt, having an impact on future modifiability and reusability when trying to extend the software, and the code would need to be refactored. As noted above, difficulties with ensuring future correctness could also arise. As new methods and functionality is added into the software, new test cases and test data will need to be developed. [Weiss mentions mainainability related issues under pain points. —SS]

8.9. Quality of Documentation

Interviewees commented that documentation is important and that its quality could be improved. As already noted, there is often no time or funding for maintaining quality documentation for software that is rarely used outside of the development team. Furthermore, the documentation generally only provides a shallow overview of the underlying CFD theory. Users would be well advised to already be familiar with these topics, or they should spend significant time referencing theory resources. The documentation instead generally focuses on explaining how to use the software. It is of course not feasible for package documentation to address the underlying physics topics in detail, so it is advised that the package documentation links to resources that better explain the underlying theory. Sometimes, frequently asked questions about the underlying theory are answered in the documentation. OpenLB has an artifact for such questions. [Weise mentions documentation. —SS]

The developer comments emphasized an importance on source code, while documentation seems to be of secondary importance. It must be stressed that improving documentation could benefit development and help eliminate some of the developer concerns that were raised. The use of automatic document generation tools that capture scientific and computing knowledge, and transform it into software artifacts, is advised. Drasil is an automatic document generation tool that is further discussed in Section 12.1.

9. LBM Developer Identified Concerns

Answers RQ9. This section presents qualitative findings on software qualities as gleaned from interviewing 4 developers. The full interview questions are found in Smith et al. (2021).

9.1. Correctness and Verifiability

Interviews with developers confirmed that these software packages are developed by domain experts with backgrounds in physics, mathematics and mechanical engineering. [Could point to surveys that have shown the same thing. —SS] It was noted in these interviews that some of the developers do not have formal software engineering education. [Can point to other studies to support this. —SS] Some of the development teams include computer scientists. Despite a lack of visible domain documentation and a resulting lower surface correctness and verifiability score, it is clear that some of the software packages were developed by teams with significant domain knowledge.

Interviews suggested a more frequent use of both unit testing and continuous integration in the development processes than what was observed from the initial survey. For example, OpenLB, pyLBM, and TCLB use such methods during development, despite this not being explicitly clear from an analysis of the material available online. [Add this to threats to validity. —SS] The correctness and verifiability of such packages is not measured well using surface analysis.

Several developers alluded to difficulty with testing the correctness of large numbers of features, and some even manually tested program output, as opposed to using automated testing. The use of well defined unit testing tools could decrease the time spent testing some feature. [link this to recommendations —SS]

9.2. Usability

Interviews with developers revealed several usability issues. Some users have misunderstood the boundaries of LBM and CFD, and have combined or applied methods that are not physically sound. Sometimes users have applied LBM to poorly defined or inappropriate fluid dynamics problems. For example, they may wish to model flow through or around a structure despite having limited information about the structure or its environment, and having little previous knowledge of CFD. The users do not realize the limitations of the methods, of the software, and do not understand the requirements to properly model a problem with the software. As the developer of TCLB noted, such software packages are not designed to be used “out of the box” in a plug and play fashion, and it could take months or more to set up the CFD problems correctly. Developers of some software packages, including ESPResSo, mitigated this by editing the source code to prevent users from “combining methods that are not physically sound together”, and by updating the documentation to better inform users of LBM limitations, and of the requirements to properly model appropriate problems, including what algorithms and parameters to use. [Could point to this for recommendations related to documentation. Could point to this for a generative approach that won’t allow inadmissible combinations, or at least includes an explicit warning. —SS]

Usability concerns were also raised for installation, understanding how to maneuver the interfaces and understanding how to set up and run models. These issues are addressed by various user support models, including frequently asked questions sections on the software package websites, user guides, and hardware and software requirements specifications. [Could point to there being room for improvement here, especially for requirements specification. —SS]

One software package (ESPResSo) changed some of its scripting language to Python to make it more usable. The developer commented that this was “the biggest step in terms of usability over the years”, further commenting that “most people in the field know [Python]” and that “it’s easy to learn”. [A generative approach facilitates switching to whatever language is currently popular. —SS]

9.3. Maintainability

Although not always visible from the publicly available artifacts, interviews with developers revealed that most projects have a defined process for accepting contributions from team members. The packages rarely get contributions from outside developers, but the process would be similar as for the aforementioned group. In general, contributions are made through GitHub, and are then reviewed and pulled by lead developers, often with consultation with a group of core developers depending on the organizational model. Continuous integration is part of the process for some packages. [Recommendation to increase. Successful projects use it, a reasonable next step for projects that have not yet incorporated CI. —SS]

Some developers noted that their software package does not have well defined contributing guide in the repository, but it might be a good idea to add one in the near future. They would be happy to see contribution from outside of their organization, but currently this does not happen. [A comment was made at the recent DOE workshop that you get new users through documentation and tutorials. Maybe highlight the example of Musubi? Should trace back to this for the recommendations part. —SS]

Furthermore, maintainability has been addressed by increasing source code modularity, reducing duplicate information, and improving abstraction by developing well defined interfaces. This was noted by the developers of ESPResSo and pyLBM. Several software packages have had sections of their code base redeveloped with languages that the developers felt are more understandable and readable, and that are better supported, such as Python. Data structures have also been redeveloped and storage has been improved. A 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". [Highlights the advantage of design. Other projects can look to these examples. Maybe cite the Anshu paper about Flash? Dune? and point to the advantages of design. —SS]

The developer of TCLB mentioned that their package had two sets of code, for executing the models on the CPU and GPU, and that maintenance was decreased by introducing macros, a practice which became a common part of the development process.

9.4. Modifiability

Software packages were not quantitatively measured for modifiability. In this project we asked developers to comment on modifiability when we interviewed them. Specifically, we asked if ease of future changes to the system, modules, and code blocks was considered when designing the software. We also asked if any measures had been taken to ensure the ease of future changes. All of the developers that were interviewed noted that the ease of future changes was considered and that measures to ensure it had been taken, including requiring the separation of software components in the source code architecture. [A good approach for other developers to adopt - part of recommendations. Trace back to this comment. —SS] A high degree of code modularity and abstraction was noted by developers as a measure to ensure the ease of future changes. This can be ensured by separating components and hiding information behind well defined interfaces. The developer of ESPResSo also noted that some of the code base was transitioned from C to C++, which could ease modifiability of that software package. 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". Furthermore, the package was designed with flexible data structures and storage in mind.

Some software packages, like Palabos, provide validation benchmarks for their core fundamental algorithmic ingredients (Latt et al., 2021). The stated intent of these benchmarks is to showcase the validity and usefulness of the package to stimulate the development of third-user extensions. The Palabos package identifies as a development framework for modeling problems in various CFD areas. [Add validation benchmarks as a "best practice" used by Palabos - others could emmulate. Goal of this exercise is for people in the community to learn from each other, and potentially from our advice too. —SS]

Developers were asked to comment on the obstacles in their development process. The developer of ESPResSo noted that a lot of their source code had been written with a specific application in mind, and that there is too much coupling between components. Addressing this issue, possibly by making the design review process more transparent (Section 9.7), would help with code modifiability and reusability. [Traceback for recommendations. —SS]

9.5. Understandability

Software developers noted that they believe users have generally found their packages to be understandable. [Confusion between understandability and usability - threat to validity example. —SS] The interviewed developer of ESPResSo commented that some users have attempted to run physically incompatible LBM methods, and the solution was to edit the code to prevent such combinations, as well as to update the documentation to prevent misunderstanding the methods. Similarly, a developer of pyLBM noted that some users had issues setting up parameters for LBM schemes. The solution to this was to update the interface where these parameters are set, as well as to add functionality to test the stability of the parameters. A developer of OpenLB noted that some users lack the background knowledge to easily model fluid dynamics problems using their software. A frequently asked questions section was added to their package website to help users find answers to common questions. The package also has detailed documentation, including guides and usage requirements specification, to better help users understand the software. [Peter - the discussion here is more about user usability than the understandability of the code for developers. Did any of the developers discuss developer understandability? The confusion between understandability and usability was also a confusion during the interviews Ao conducted. —SS] [The developer of ESPResSo did comment on developer understandability, specifically for developers that are/were part of their peer review process. He commented that code architecture is documented, and that code is made to be readable, and that these help peer reviews of the code. —PM]

[I moved this from the visibility section, since it is more about understandability. —SS] Developers were also asked how documentation fits into their development process. Several developers noted that developer documentation plays an important role in familiarizing potential contributors to the software system architecture. Without the guidance that the documentation provides it would be unlikely that contributions would pass the peer review process.

9.6. Traceability

Software packages were not quantitatively measured for traceability. In this project we asked developers to comment on traceability, specifically on their software package's documentation and how it fit into their development process.

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 had been put into the documentation. They further commented that they want to lower the entry barrier for new developers, and because of that their package has a considerable amount of developer documentation. This documentation informs developers on how to get started, and orients them to the artifacts, source code, and system architecture, as well as how the software package build system works, and how the coupling between the simulation engine and the interface works. [Peter - these points about developer documentation seem like they would fit better under understandability, don't you think? —SS] [After reviewing the definitions of the two qualities, I think you are right. I think that the point presented in the first sentence of this paragraph belongs here in traceability (but could be reworded to note that ESPResSo also has detailed change logs in their docs - change logs are also available for Palabos and waLBerla), but the rest of the paragraph can be moved to understandability. —PM] [If ESPresSo scored high (I think it did), we can use that to further promote the value of documentation. Also a threat to validity is that we have implicitly assumed documentation is valuable when measuring quality. —SS]

Developers noted the importance of documentation for both the users and developers of their software. New features are always added to the documentation. The developers use documentation to stay up to date on the status of the software package, and to help expand features, like computational models or algorithms. This is necessary so that the coding standard for these models is kept consistent for new developers. [Peter - this also sounds like discussion that is relevant for understandability. —SS] [The points here sound like they could work for both. The definition of traceability notes that 'This quality refers to keeping track of information as it changes forms or relates between artifacts'. So the point 'New features are always added to the documentation' could be reworded to fit that part of the definition of traceability. As noted above, this is similar to the point about some packages having including detailed change logs. Same with 'The developers use documentation to stay up to date on the status of the software package, and to help expand features'. I agree that at least some of this paragraph could be in understandability, especially the first and second sentences. —PM]

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 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. Sometimes it is a last priority for the developers. Finding ways to hasten updating documentation would increase the frequency of such updates and benefit both users and developers. [Same thing has come up in papers. I'll try to find a reference for this. This is also something we can point to for recommendations related to code generation. —SS] [Peter - this also sounds like understandability related discussion? I wonder if we should drop Traceability as a quality that we assessed? There doesn't seem to be much content specific to traceability. —SS] [Based on the comments I just added above, I think we can keep at least a small part of what is here in traceability, rewording parts to highlight the importance of change logs. Some of the content in this section should indeed be moved to understandability. —PM] The developer of OpenLB noted the use of documentation generators like Doxygen. It would be advisable for more projects to use such automatic document generation tools, since some projects do not do this. [We can refer back to this for the recommendations. —SS]

9.7. Visibility and Transparency

For the developers interviewed, visibility is not currently a priority. Visibility is not particularly necessary because the teams are fairly small. Although they accept outside contributors, these are rare and the teams remain tight-knit. Most of the teams interviewed have all members at the same institution, although one team is spread out internationally. [Peter, can you say which team is international, so that we can add this detail to the paper? —SS] [The interviewed developer of OpenLB noted that there were tasks for local and international developers when discussing their software development model. He also noted that they have frequent local meetings and annual international meetings. —PM] For teams of only a few developers, new features or major changes are discussed with the entire team. Projects with more than a few developers have lead developer roles. These lead developers review potential additions to the software.

Although visibility is not a priority, positive steps are being undertaken. For instance, all interviewed teams used GitHub, with its visibility enhancing features of branching, issue tracking, insights on contributors, the blame feature, wikis, pull requests, etc. Continuous integration, as used by ESPResSo, makes projects even more visible, since contributors can explicitly see the bar that must be crossed for code to be incorporated into the master branch. ESPResSo also uses peer review, which improves visibility by explicitly asking for feedback from other developers.

Previously mentioned concerns with maintainability, modifiability, and understandability could be improved by further improving visibility. For future sustainability, projects will likely have to expand their development teams by recruiting more external developers. One component of recruiting developers is transparency in the development process, including the process for making a contribution. [in recommendations could point out what teams could do to get more external contributors, and potentially a longer life for their project. —SS]

9.8. Reproducibility

Software packages were not quantitatively measured for reproducibility. Instead, developers were asked if they had concerns that their computational results would not be reproducible in the future, and if they had taken any steps to ensure reproducibility. The developers all agreed that reproducibility is important, but they differ in how far they go to support reproducibility. ESPResSo does the most to support reproducibility. For instance, they compare the results of their methods against manually calculated results. These comparison tests are automatically run for all source code changes. The tests are run when a pull request is opened on GitHub. Even once these tests are complete, a peer review process is undertaken before changes are fully committed to the appropriate branch. The results for all of the LBM schemes on the software package development branch are also frequently compared for correctness, ensuring that the system output reflects the expected output.

For other teams, the priority is verifying the mathematical foundations of the models, but the output of the software package is not compared to other output. Depending on the package and how it outputs solutions, this may not be practical or feasible. A correct output may not be exactly reproducible, as it may be dependent on a probability distribution, so strictly comparing results may not be appropriate. [Ideas for dealing with reproducibility in these situations will be explored in the recommendations section (like properties of the solution, such as conservation of mass). —SS]

For the current project, we did not assess the stronger form of "reproducibility" known as replicability. A software project is considered replicable if the documentation alone (without the code) is sufficiently precise and complete to allow a third-part to re-obtain the results of the original code (Benureau and Rougier, 2017). Replicability is difficult to achieve with current documentation practices. This question should be considered in the next iteration of state of the practice assessments.

9.9. *Unambiguity*

Software packages were not quantitatively measured for unambiguity. We asked developers if they thought that the current documentation can clearly convey all necessary knowledge to the users, and if they had taken any steps to ensure clarity. The developer of ESPResSo noted that their documentation was meant for users that are already familiar with the underlying physics and CFD methods. These concepts are not explained in detail within the documentation. Users should acquire this knowledge from suitable external sources. The documentation focuses on how to technically use the software package, and includes a user guide and tutorial walk through of how to set up and run a simulation. With this in mind, the developer believes that their documentation is in reasonable shape for users with a minimum knowledge of the underlying physics. If new users have technical questions these can be addressed in further revisions of the documentation. New developers should find that the documentation is reasonably clear and useful. Information that is missing, like detailed explanations of dependencies, is referenced in the documents.

The developer of pyLBM also noted that their documentation was in reasonable shape, but that they "need more [user] feedback to improve [it]". They also noted that they believe a lack of knowledge of the underlying physics and CFD concepts can be an issue for some users. This information can be referenced in the documentation, but it is not something that the documentation needs to detail.

10. Lessons from LBM Developers

Answers RQ10. Recommendations for Addressing Developer Concerns. This section answers the research questions listed in Section 1 using the quantitative data presented in Section 3, qualitative data presented in Section **??**, additional data from software package repositories and artifacts, and domain expert feedback.

10.1. *Common Practices*

The practices that arose frequently. Compare and contrast to what is done with research software in general. The following points regarding software quality should be considered when developing LBM software packages. These points are based on developer interviews, SCS literature, what was found to have worked for packages that were designated as high quality in this assessment. Our recommendations are not lists of what should have been done in the past, or what should be done now; they are just suggestions for consideration in the future. [Rework this section to highlight what the top projects are doing. Point to checklists for other advice? —SS] Highlights of common practices. Put in context.

Conclusion that LBM software common practice is similar to the common practice elsewhere. Maybe point out where LBM is ahead. Use results from common artifacts, common process etc. to support conclusion. (Continuous integration, version control, README, Req etc. - point to Table 8.)

- Surface Reliability. Surface Usability. The packages should include detailed tutorials, including dependencies, expected output, and any additional supplementary documentation that may be required. This was done by many packages, including waLBerla, Palabos, MechSys, LUMA, and pyLBM. [Comment from ASCR attendee that success with open source requires good tutorials and documentation - is this published anywhere? I think I can find this recommendation in other papers. —SS]

- Surface Usability. Unambiguity. Surface Understandability. Provide a detailed user manual. It should identify elements of user interfaces, and identify all requirements to model a system. Thirteen packages have a user manual. ESPResSo has a well detailed manual. [Include this with tutorial? Find papers that recommend this kind of documentation? —SS]

- Surface Usability. Include user hardware and software requirements documentation. Hardware requirements are rarely listed in the software packages that were assessed. Only those offering GPU processing (ESPResSo, lbmpy, lettuce, pyLBM, Sailfish, TCLB) mentioned any hardware requirements. On the other hand, some sort of software requirements were available for all packages, even if only some dependencies or a compatible operating system were mentioned. [This is a key part of reproducibility. —SS]

- Not much of a process - find papers that support this - papers recommend "Define clear and transparent contribution, governance and communication processes" JimenezEtAl2017

10.2. Best Practices

practices by the best measured.

- The top projects provide great examples of achieving installability, by taking such steps as: providing complete installation instructions, writing instructions as if the user does not have any of the dependencies installed, automation of the installation process (for instance using make), including descriptive error message, and including steps for uninstallation.
- Surface Correctness and Verifiability. Make public (on package website or GitHub) the requirements specification document, or explicitly reference the domain theory that the software is designed from. This was done by several top ranked packages, including ESPResSo, Ludwig, LUMA, and OpenLB.
- Surface Correctness and Verifiability. Ensure the above information is easy to find. Consider adding it to the user manual. The information is easy to find in most top ranked packages.
- Surface Correctness and Verifiability. Development teams should include both domain experts and experienced software developers. This suggestion is based on developer comments.
- Surface Correctness and Verifiability. Traceability. Visibility and Transparency. Use and make public (on package website or GitHub) detailed documentation. Consider using automatic document generation tools like Doxygen, Drasil, or Sphinx. This was done by all of the top five ranked packages.
- Surface Correctness and Verifiability. Provide detailed tutorials that include expected output, like the waLBerla tutorials.
- Surface Correctness and Verifiability. Use unit tests during development and make them public (on package website or GitHub). This was available for Ludwig.
- Surface Correctness and Verifiability. Modularize the source code, separate components, hide information behind well defined interfaces. This is suggested in SCS literature, and in developer comments. [one of Wilson's best practices. —SS]
- Surface Understandability. Maintainability. Modifiability. Reusability. Modularize the source code, separate components, hide information behind well defined interfaces. This is suggested in SCS literature, and in developer comments. [add citation - Wilson mentions - this points seems repetitive —SS]
- Surface Correctness and Verifiability. Use continuous integration tools (Bamboo, Jenkins, and Travis CI) and processes during development. This was done by top ranked packages ESPResSo and Ludwig. [Common in LBM - more common than in other domains —SS]
- Surface Usability. State appropriate fluid dynamics problems that the software is designed to model in the documentation, and explicitly state the limits of the software. This is done in the ESPResSo user guide.
- Surface Usability. Provide documentation that details the background theory information, or provide a reference to such information. This was done by several top ranked packages, including ESPResSo, Ludwig, LUMA, and OpenLB.
- Surface Usability. Identify expected user characteristics. LIMBES, Ludwig, laboetie, and Palabos did this. The importance of specifying user characteristics is discussed in (Smith et al., 2007).
- Surface Usability. Keep all documentation in one location. This was done by top ranked packages.

- Surface Usability. Maintain a user support model (Git, email, forum, FAQ). This was done by all of the top five ranked packages.

- Surface Usability. If possible, consider using popular user-friendly software languages like Python. Especially consider this for parts of the source code that is likely to be modified or reviewed by users. ESPResSo and Sailfish use Python to shorten development time and improve usability.

- Maintainability. Include version numbers and release notes for all major source code and artifact releases. The top five ranked software packages include release notes.

- Maintainability. Have a defined process for accepting contributions, and make public documentation for making contributions to the project. The top five ranked software packages include information on how to contribute.

- Maintainability. Use an issue tracker (Git, email, SourceForge, other) to manage bugs and changes. Issues should be regularly reviewed and closed. All but three (laboetie, lettuce, Sailfish) of the software packages that use Git have most of their issues closed.

- Maintainability. Use a version control system (GitHub, CVS). Four (ESPREsSo, Ludwig, LUMA, Palabos) of the top five ranked packages use a version control system.

- Maintainability. Source code needs to be well commented. Typically, more than 10 percent of LBM package source code is comments, as presented in Table 4. Four of the top five overall ranked packages (Ludwig, ESPResSo, Palabos, OpenLB) have about 20 percent of their source code as comments.

- Maintainability. If possible, consider using popular user-friendly software languages like Python. Especially consider this for parts of the source code that is likely to be modified or reviewed by users. ESPResSo and Sailfish use Python to address several software qualities, including maintainability.

- Modifiability. Consider flexibility of data structures and data storage in the design stage. The package pyLBM redeveloped data structures to ease future changes.

- Reusability. Provide API documentation, if applicable. Only one (ESPResSo) of the top five ranked packages provided API documentation.

- Surface Understandability. Adopt a coding standard and document it in artifacts, including some examples. A coding standard needs to be part of continuous integration. Only LUMA, HemeLB, and Musubi explicitly identify coding standards.

- Surface Understandability. Add meaningful comments. Indicate what is being done in each section of source code. All of the top ranked packages have meaningful comments in their source code.

- Surface Understandability. State appropriate fluid dynamics problems that the software is designed to model in the documentation, and explicitly state the limits of the software. This is done in the ESPResSo user guide.

- Surface Understandability. Consider adding a FAQ section to the documentation. This helped resolve some usability issues for OpenLB.

- Traceability. Provide a developer's guide to help orient developers to the artifacts, source code, and system architecture so that they can better document changes. Top ranked package OpenLB has a developer guide. [How is this related to traceability? —SS]

- Visibility and Transparency. Summarize the development process that is used. Provide information on how new users can contribute. Identify the development model by name (waterfall, agile, etc.), if appropriate. Seven of the software packages have some artifacts outlining the general development process. Eleven packages have information on how to contribute.

- Visibility and Transparency. Identify the development environment. Eight of the 24 software packages identify the development environment.

- Visibility and Transparency. Include notes with all releases. Nine of the packages include release notes.
- Visibility and Transparency. Communicate all changes within the development team. This suggestion is based on developer comments.
- Visibility and Transparency. Use continuous integration processes and tools (Bamboo, Jenkins, and Travis CI). This was done by top ranked packages ESPResSo and Ludwig.
- Visibility and Transparency. Consider peer review processes to assess contributions and ensure the tracking of information. High ranked package ESPResSo uses a peer review process for contributions.
- Reproducibility. Test output against automatically calculated or known correct results. High ranked package ESPResSo does this.
- Reproducibility. Automate the testing of output. Run tests after all code changes. High ranked package ESPResSo does this.
- Reproducibility. Consider peer review processes and task based inspection to assess contributions. High ranked package ESPResSo does this.
- Unambiguity. State appropriate fluid dynamics problems that the software is designed to model in the documentation, and explicitly state the limits of the software. This is done in the ESPResSo user guide.
- Unambiguity. The documentation should either explain the underlying CFD theories or provide a reference to appropriate resources. This was done by several top ranked packages, including ESPResSo, Ludwig, LUMA, and OpenLB.
- Unambiguity. Consider asking users for feedback on the documentation. The developer of pyLBM noted that such feedback would be appreciated.

practices mentioned in literature, but not practiced (if any). Start with the practices that aren't derived from developer interviews.

- Surface Correctness and Verifiability. Use a requirements specification document. This is suggested in SCS literature, and several of the top ranked packages had such a document or reference to theory manuals. A potential template is presented in (Smith and Lai, 2005).
- Surface Usability. Do not duplicate artifact information. This is suggested as good software development practice. Duplicate information is difficult to maintain.
- Maintainability. Visibility and Transparency. Reproducibility. Keep artifacts updated. This is suggested as good software maintenance practice.
- Maintainability. Eliminate code duplication. This is suggested as good software development practice. Duplicate code is difficult to maintain. [No support for this one or citation, either find or remove it. —SS]
- Modifiability. Consider future source code modifiability as early as the design stage of development. This is suggested as good software development practice. [citation, support, or remove. —SS]
- Reusability. Document module interfaces in design and developer documentation. This is suggested as good software development practice. [citation or support —SS]
- Traceability. Update all relevant documentation when a change to the software is made. This is suggested as good software development practice. [support, cite or remove —SS]
- Unambiguity. Documentation must state all technical requirements and software dependencies. A missing dependency was a frequent cause of fault conditions during this assessment.
- Unambiguity. Provide a table of symbols in the developer's guide that maps to names used in the source code. This is suggested as good software development practice.

10.3. Future Practices

Answers RQ11. practices for the future.

"unique challenges of scientific research (e.g., the frequent and unforeseen changes in requirements ...)" (Pinto et al., 2018). We can point out that in the future this viewpoint could change and why. Not really a unique challenge. Cite Smith 2016?

11. Threats To Validity

This section examines potential threats to the validity of this state of the practice assessment. These can be categorized into methodology and data collection issues. The goal of this assessment isn't to rank the software, but to use the ranking exercise as a means to understand the state of the practice of LBM software development.

The measures listed in our measurement template may not be broad 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 faults could be introduced, but could require a large investment of time to develop, and might not be a fair measure for all packages. Similarly, reusability is assessed along 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. Furthermore, the measurement of understandability relies on 10 random source code files. It is possible that the 10 files that were chosen to represent a software package may not be a good representation of the understandability of that package.

Regarding data collection, a risk to the validity of this assessment is missing or incorrect data. Some software package 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 **??**.

Some pertinent data may not have been specified in public artifacts, or may be obscure within an artifact or web-page. The use of unit testing and continuous integration was mentioned in the artifacts of only three (ESPResSo, Ludwig, Musubi) packages. However, interviews suggested a more frequent use of both unit testing and continuous integration in the development processes than what was observed from the initial survey of the artifacts. For example, OpenLB, pyLBM, and TCLB use such methods during development despite this not being explicitly clear 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 SCS is not a trivial endeavor. It is likely that many packages have such documentation but did not make it public, and due to this the measured data is not a true reflection of software package quality.

[There are more threats to validity. Brainstorm. Watch for them while editing. Look at Ao paper. —SS]

12. Conclusion

We analyzed the state of the practice of software development in the Lattice Boltzmann Methods software domain by quantitatively and qualitatively measuring, and comparing, 24 software packages along quality attributes. The software qualities that were assessed in this report are listed in Section 2.1. A methodology for assessing the state of the practice of software development in SCS domains was presented. Domain packages were assessed to answer the software development related research questions listed in Section 1 to understand how software quality is impacted by software development choices, including principles, processes, and tools. Software developers were interviewed to identify development pain points, and to identify how software quality is ensured. Quantitative data was used to rank the software packages using the AHP. The ranking designations were compared with rankings from the software development community, and we found that many of our top 10 ranked packages are ranked highly by a domain expert. Recommendations for improving software along quality metrics were made, and highlights are presented in Section 12.1 of this conclusion. The findings of this report can be used to guide future development of SCS, specifically along quality attributes, and to reduce software quality failures.

[More on the lessons to learn from the top packages. The comparison to common practice - is LBM ahead or behind? More on best practices comparison. Future practices. —SS]

We understand that software packages vary in goals, developers, funding, and other aspects. Our goal was to highlight software quality successes through an evaluation of the entire domain software family. Furthermore, threats to the validity of the findings are highlighted in Section 11.

Recommendations for future state of the practice assessments are made in Section 12.2.

12.1. Highlighted Recommendations

The following recommendations improve software quality along multiple attributes, and provide the greatest return on investment:

- Provide a detailed user manual. It should identify elements of user interfaces, and identify all requirements to model a system. Thirteen packages have a user manual. ESPResSo has a well detailed manual.
- State appropriate fluid dynamics problems that the software is designed to model in the documentation, and explicitly state the limits of the software. This is done in the ESPResSo user guide.
- Include detailed tutorials, including dependencies, expected output, and any additional supplementary documentation that may be required. This was done by many packages, including waLBerla, Palabos, MechSys, LUMA, and pyLBM.
- Keep all documentation in one location. This was done by top ranked packages.
- If possible, consider using popular user-friendly software languages like Python. Especially consider this for parts of the source code that is likely to be modified or reviewed by users. ESPResSo and Sailfish use Python to address several software qualities, including maintainability, modifiability and usability.
- Modularize the source code, separate components, hide information behind well defined interfaces. This is suggested in SCS literature, and in developer comments.
- Include descriptive error messages where appropriate. This was done by most packages that encountered a fault.
- Summarize the development process that is used. Provide information on how new users can contribute. Seven of the software packages have some artifacts outlining the general development process. Eleven packages have information on how to contribute.
- Consider peer review processes and task based inspection to assess contributions. High ranked package ESPResSo does this.
- Use continuous integration tools (Bamboo, Jenkins, and Travis CI) and processes during development. This was done by top ranked packages ESPResSo and Ludwig.
- Use project management tools, including change and version control tools (GitHub, GitLab, CVS), collaboration tools (GitHub, GitLab), and document generation tools (Doxygen, Drasil, Sphinx). This was done by all of the top five ranked packages.

Ensuring software quality can take significant time. An inability to develop high quality documentation due to time constraints could be mitigated in the future by the use of automatic document generation tools like Drasil. The Drasil Framework consists of a collection of Domain Specific Languages (DSL) for capturing scientific documents, structures, and computing knowledge, and then transforming this knowledge into relevant software artifacts without having to manually duplicate knowledge into multiple artifacts (Zhao, 2018). [Yuzhi is not the best reference to use for the Drasil work. —SS]

12.2. Future State Of The Practice Assessments

As mentioned in Section 11, the measures listed in our measurement template may not be broad enough to accurately capture some qualities. Adding or extending measures is worth considering. For example, usability experiments and performance benchmarks could be incorporated into the assessment.

Section **??** noted that developers were not asked to comment on the reproducibility of their source code from their requirements specifications and design documentation. Adding this question to the interview guide should be considered in the next iteration of state of the practice assessments. Furthermore, as mentioned in Section 11, it was found that some pertinent information was not specified in public artifacts. The use of unit testing and continuous integration by several packages (OpenLB, pyLBM, TCLB) was only discovered during interviews with developers. Adding further questions to the interview guide regarding the measures that are on the measurement template could reduce instances of incorrect data being collected. 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. [clarify the distinction between usability and understandability. —SS]

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`

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.

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.

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`

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.

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.

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. 1991. *Fundamentals of software engineering*. Prentice Hall PTR.

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

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. 2014a. Complex fluid simulations with the parallel tree-based Lattice Boltzmann solver Musubi. *Journal of Computational Science* 5, 5 (2014), 784–794. https://doi.org/10.1016/j.jocs.2013.11.001

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

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.

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

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.

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.

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

Daniel S Katz, Daina Bouquin, Neil P Chue Hong, Jessica Hausman, Catherine Jones, Daniel Chivvis, Tim Clark, Mercè Crosas, Stephan Druskat, Martin Fenner, et al. 2019. Software citation implementation challenges. *arXiv preprint arXiv:1905.08674* (2019).

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

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.

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.

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.

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/

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

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

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. 2007. End-user software engineering and professional end-user developers. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-

Leibniz-Zentrum fr Informatik.
Arfon M Smith, Daniel S Katz, and Kyle E Niemeyer. 2016a. Software citation principles. *PeerJ Computer Science* 2 (2016), e86.
Spencer Smith, Jacques Carette, Olu Owojaiye, Peter Michalski, and Ao Dong. [n.d.]. Quality Definitions of Qualities. `https://github.com/smiths/AIMSS/blob/master/StateOfPractice/QDefOfQualities/QDefOfQualities.pdf`
Spencer Smith, Lei Lai, and Ridha Khedri. 2007. Requirements analysis for engineering computation: A systematic approach for improving reliability. *Reliable Computing* 13, 1 (2007), 83–107.
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, 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 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*, Vol. 5. Citeseer, 107–121.
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. 2018a. State of the Practice for GIS Software. `https://arxiv.org/abs/1802.03422`.
W. Spencer Smith, Yue Sun, and Jacques Carette. 2018b. 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. 2018c. Seismology Software: State of the Practice. *Journal of Seismology* 22, 3 (May 2018), 755–788.
Ian Sommerville. 2011. *Software Engineering 9*. Pearson Education.
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.
Wiegers. 2003. *Software Requirements, 2e*. Microsoft Press.
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`
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`
Yuzhi Zhao. 2018. Automated Knowledge Extraction Based On A Scientific Computing Software Documentation Generation Framework.