State of the Practice of Lattice Boltzmann Method Software Development

Peter Michalski

May 25, 2021

Abstract

This project analyzed the state of the practice for software development in the Lattice Boltzmann Method domain by quantitatively and qualitatively measuring 23 software packages along quality attributes...

Contents

List of Figures

List of Tables

1 Reference Material

1.1 Software Engineering Related Definitions and Acronyms

AHP Analytical Hierarchy Process

Commonality: A requirement or goal common to all family members.

Goal: Goals capture, at different levels of abstraction, the various objectives the system under consideration should achieve. (Van Lamsweerde, 2001)

Program Family: A set of programs that are analyzed and designed together starting from the initial stages of the software development life-cycle.

Requirements: A software requirement is: *i*) a condition or capability needed by a user to solve a problem or achieve an objective; *ii*) a condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document; or, *iii*) a documented representation of a condition or capability as in the above two definitions. (Thayer and Dorfman, 2000)

Variability: A requirement or goal that varies between family members.

1.2 Lattice Boltzmann Related Definitions and Acronyms

1D 1-Dimensional

2D: 2-Dimensional

3D: 3-Dimensional

BGK: Bhatnagar-Gross-Krook

CPU: central processing unit

GPU: graphics processing unit

MRT: multi-relaxation-time

SRT: single-relaxation-time

TRT: two-relaxation-time

LBM: Lattice Boltzmann Methods

LBS: Lattice Boltzmann Solvers

Velocity Directions: The number of links connecting to each lattice node in the chosen model from neighbouring nodes. All nodes in a chosen lattice model will have the same number of links. A single link will connect between two adjacent nodes.

2 Introduction - DONE SECTION 1st DRAFT

This report analyzes the development of software packages that use Lattice Boltzmann Methods (LBM), a class of computational fluid dynamics methods, for fluid simulation. The analysis involves taking qualitative and qualitative measures along quality attributes. The goal is to analyze the current state of development of Scientific Computing Software (SCS) in this domain to provide insight into its best practices and offer guidance for future development.

In this report red text denotes a linked sections of this document. Cyan text denotes a URL link. Blue text denotes a citation.

2.1 Purpose of Report

This document constitutes the author's M.Eng. report in partial fulfillment of the the degree of Master of Engineering. It is to be submitted to the Department of Computing and Software in McMaster University's Faculty of Engineer and is intended for review by the examination committee. In this context the purposes of the report are the following:

- Describe the state of the practice of SCS project origin, motivations, and goals
- Describe LBM software structure and goals
- Report on the measure of a subset of LBM software packages along quality metrics
- Evaluate the state of the practice of LBM software development along quality metrics
- Suggest future work for improving LBM software along quality metrics
- Suggest future work for improving state of the practice of SCS assessments

2.2 Motivation of Report

The purpose of state of the practice assessments is to understand how software quality is impacted by software development choices, including principles, processes, and tools, within SCS communities. This knowledge can be used to guide future development of SCS, specifically along quality attributes, and reduce software quality failures. This work reports on the state of the practice of LBM software development and makes suggestions on improving the quality of software in this domain.

This assessment of the state of the practice of LBM software development builds off of prior work on assessing the state of research software development. The previous state of the practice assessments include domains such as Geographic Information Systems (Smith et al., 2018a), Mesh Generators (Smith et al., 2016), Oceanographic Software (Smith et al., 2015), Seismology software (Smith et al., 2018c), and statistical software for psychology (Smith et al., 2018b).

In the course of this assessment we updated the methodology that was used during previous assessments. Details of the new methodology are found here. The previous set of research questions were critically assessed and modified. The updated questions are listed in Section 9.1. In this re-boot we collected more quantitative and qualitative data, focused on software quality attributes, by adding more measures, including collecting empirical data and interviewing developers. We have also added a domain analysis to better characterize the functionality provided by the software, and have leveraged the expertise of a domain expert. As in past assessments, the collected data was combined to rank the software using the Analytic Hierarchy Process (AHP). The domain expert was consulted to verify the ordering.

2.3 Scope of Report

This report analyzes a filtered set of LBM software packages along quantitatively and qualitatively measures. Many of these measures are captured using surface measurements, which can be categorized as initial and easy to capture measurements, of the underlying quality and they may not represent the true quality of the software. These surface measurements are taken due to the desire to apply the same measurement, with reasonable effort, along all software packages despite technical and functional variabilities in the set of software packages. The measures are used to answer the research questions found in Section 9.1. The measured software packages are open source, and it is recommended to keep this in mind when considering the recommendations of this report. Practices surrounding close source software development may differ.

The report addresses what was done during the development of the software packages to address the quality attributes listed in Section 5.2. It then provides general guidance on how to improve these qualities when developing LBM software. It does not make suggestions on what should have been done or should be done for any one specific package. Best practices may at times differ among software packages.

2.4 Organization of Report

The first part of this document includes the Table of Contents, List of Figures, and List of Tables. This is followed by Reference Material including a subsections outlining Software Engineering Related Definitions and Acronyms and Lattice Boltzmann Related Definitions and Acronyms.

The rest of core of the report is organized as follows:

- **Introduction** to the report, including its purpose, motivation, scope, and organization.

- **Background** of Scientific Computing Software, including software qualities and software families.
- LBM software **Domain Analysis**.
- **Methodology** of this state of the practice assessment, including the steps of the methodology, definitions of software qualities and how they are applied in this assessment, and an overview of candidate software selection, empirical measures, and the analytical hierarchy process.
- Quantitative and qualitative **Results**.
- An analysis of the results and **Answers To Research Questions**.
- **Conclusion** to the report including comments on future state of the practice assessments.
- **Appendix**: This section includes our Research Questions, the Measurement Template, Grading Template, Ethics Approval, and Developer Interview Questions.

3 Background - DONE SECTION 1st Draft

Scientific computing software (SCS) is defined as the use of computer tools to analyze or simulate mathematical models of real world systems (Smith, 2006). Given the importance of such software, scientists and engineers desire methods and tools to sustainably develop high quality software, which has led to the formation of groups like the Software Sustainability Institute (SEI) and Better Scientific Software (BSS). The purpose of this state of the practice assessment is to understand how software quality in the LBM SCS family is impacted by software development choices, including principles, processes, and tools. A software family is defined by Parnas (1976) as a set of programs whose common properties are so extensive that it is advantageous to study the common properties of the programs before analyzing individual members. The common properties of the LBM family of related programs are listed in the a domain analysis in Section 4. The software qualities that are assessed in this report are listed in Section 5.2. Ensuring these qualities can take significant time, but as Smith et al. (2016) shows, SCS developers often cannot see themselves meeting the required time constraints to better develop some of the artifacts that improve software quality, like detailed documentation. Inability to develop such documentation due to time constraints could be mitigated 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).

4 Domain Analysis - DONE SECTION 1st DRAFT

4.1 Lattice Boltzmann Systems

Lattice Boltzmann Methods (LBM) are a family of fluid dynamics algorithms for simulating single-phase and multiphase fluid flows, often incorporating additional physical complexities (Chen and Doolen, 1998). They consider the behaviours of a collection of particles as a single unit at the mesoscopic scale. These methods predict the positional probability of a collection of particles moving through a lattice structure. Off the shelf (OTS) Lattice Boltzmann Solvers (LBS) allow for a range of fluid and physical model input parameters, computational parameters, and output parameters.

As LBS model fluid dynamics within a boundary using a predefined lattice structure, the methods rely on a two step calculation process. The first process is streaming, where the particles move along the lattice via links. The second process is collision, where energy and momentum is transferred among particles that collide (Bao and Meskas, 2011). There are many standardized lattice models - individual solvers within the family might only use a subset of them. The LBM uses the initial parameters of the fluid to find the probability of where along the lattice linkages a group of particles are most likely to travel. It then moves the particles into the next node, and transfers the energy and momentum if a collision occurs. Then the process repeats for the duration of the modeling instance.

4.2 Commonalities

This section lists all the common features among all the potential family members. The commonalities are organized using the following abstraction of the system, which can be used to describe all Lattice Boltzmann systems: input information, then generate the simulation and finally output the results. Section 4.2.1 describes the commonalities for the simulation step. Section 4.2.2 highlights the input information that is required for all Lattice Boltzmann

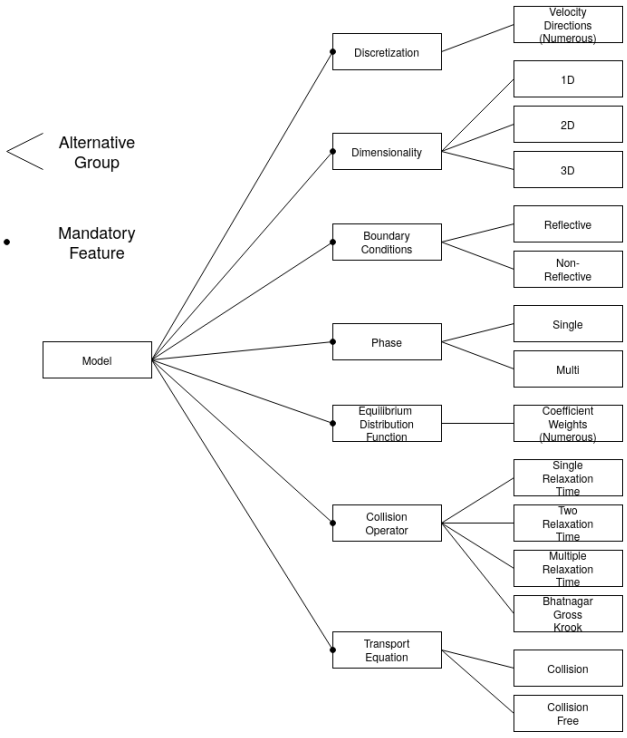

Figure 1: LBS Model Feature Diagram

systems. The next section, Section 4.2.3, shows the common features for the output of Lattice Boltzmann systems, such as the requirement that mesh information be written to files. (Although the output information could simply be written to the computers memory, in all practical applications it is desirable to have a persistent record of the output that was created.) The final section covers qualities of the system that cannot be classified as input, simulation generation or output. These commonalities are termed nonfunctional requirements of the system. For instance, all systems will have the goal that the response time to a users request is small enough to allow the user to focus on his/her problem and to maintain his/her train of thought, without being distracted by excessive waiting time. The commonality in this case is refined by a later variability because the specific requirement on the response time will depend on the intended usage of the mesh generating system.

Each commonality below uses the same structure. All of the commonalities are assigned a

unique item number, which takes the form of a natural number with the prefix "C". Following this, a description of the commonality is provided along with a list of related variabilities, which are given as hyperlinks that allow navigation of the document to the text describing the variability. Finally, each commonality ends with a summary of the history, including the date of creation and any dates of modification, along with a brief description of the modification. If necessary, a previous version of the document can be obtained by using the concurrent versioning system where the files are stored.

4.2.1 Lattice Boltzmann Method Solvers

Item Number	C1

Description	A lattice discretizes a computational domain into a finite number of points. All LBS discretize the computational domain using a regular, evenly spaced grid within a boundary.
Related Variability	V6 V7 V11

Item Number	C2

Description	All Lattice Boltzmann versions use a collision operator which concerns collisions between particles.
Related Variability	V5 V6

Item Number	C3

Description	All Lattice Boltzmann versions user a probability density function to give the probability that fluid has moved into a specific domain.
Related Variability	V5

Item Number	C4

Description	Every Lattice Boltzmann version uses an equilibrium distribution function to capture the probability distribution of particles.
Related Variability	V2 V3 V4

Item Number	C5

Description	Every Lattice Boltzmann version uses a Boltzmann transport equation to describe the statistical behaviour of a system that does not have collisions.
Related Variability	V8

4.2.2 Input

Item Number	C6

Description	The LBS require fluid, model, and boundary information for the problem.
Related Variability	V12

4.2.3 Output

Item Number	C7

Description	LBS write fluid parameter predictions to memory.
Related Variability	V13 V14

4.2.4 Nonfunctional Requirements

Item Number	C8

Description	LBS provides the precision required for the particular problems it is intended to solve.
Related Variability	V18

Item Number	C9

Description	LBS provides the accuracy required for the particular problems it is intended to solve.
Related Variability	V19

Item Number	C10

Description	The response time is small enough to allow the user to focus on their problem without being distracted by excessive waiting times.
Related Variability	V20

Item Number	C11

Description	LBS will be as portable to other operating systems as required by the users.
Related Variability	V21

Item Number	C12

Description	The use of LBS is easy and efficient.
Related Variability	V22

4.3 Variabilities

This section provides a list of characteristics that may vary among family members. As in Section 4.2, the first three subsections on variabilities are organized into the following sublists: Mesh Generation, Input and Output. The final two subsections list variabilities that can be characterized as system constraints and as nonfunctional requirements.

As for the commonalities, each variability is labelled with a unique item number. In this case the numbers are prepended with the letter "V". The other four headings provided for each variability are: Description, Related Commonality, Related Parameter and History. The related commonalities and parameters are given as a set of identifiers that respectively refer back to the previous section on commonalities or refer forward to the next section on parameters of variation.

4.3.1 Lattice Boltzmann Method Solvers

Item Number	V1

Description	LBS may use a framework for parallel processing of the model.
Related Commonality	None
Related Parameter	P1

Item Number	V2

Description	Different versions of an equilibrium distribution function can capture the probability distribution of particles.
Related Commonality	C4
Related Parameter	P2

Item Number	V3

Description	Storage patters for distribution function can vary.
Related Commonality	C4
Related Parameter	P3

Item Number	V4

Description	Coefficients for distribution function can vary.
Related Commonality	C4
Related Parameter	P4

Item Number	V5

Description	The number of dimensions in the lattice of the model can vary.
Related Commonality	C1 C3
Related Parameter	P5

Item Number	V6

Description	The number of velocity directions in the lattice of the model can vary.
Related Commonality	C1 C2
Related Parameter	P6

Item Number	V7

Description	Various collision operators can be used.
Related Commonality	C1 C2
Related Parameter	P7

Item Number	V8

Description	Various transport equations can be used to describe the statistical behaviour of the system
Related Commonality	C5
Related Parameter	P8

Item Number	V9

Description	The number of fluids allowed in the LBS.
Related Commonality	C6
Related Parameter	P9

Item Number	V10

Description	The type of fluid parameters.
Related Commonality	C6
Related Parameter	P10

Item Number	V11

Description	The boundary of the lattice can have varied conditions.
Related Commonality	C1
Related Parameter	P11

4.3.2 Input

Item Number	V12

Description	The input interface can vary between LBS.
Related Commonality	C6
Related Parameter	P12

4.3.3 Output

Item Number	V13

Description	Visual presentation of the prediction.
Related Commonality	C7
Related Parameter	P13

Item Number	V14

Description	Format of prediction information.
Related Commonality	C7
Related Parameter	P14

4.3.4 System Constraints

Item Number	V15

Description	Hardware which processes the calculations
Related Commonality	None
Related Parameter	P15

Item Number	V16

Description	Operating systems on which LBS runs.
Related Commonality	None
Related Parameter	P16

Item Number	V17

Description	Amount of storage and memory needed for the LBS.
Related Commonality	None
Related Parameter	P17

4.3.5 Nonfunctional Requirements

Item Number	V18

Description	The precision needed for each input and output.
Related Commonality	C8
Related Parameter	P18

Item Number	V19

Description	The required accuracy for the output.
Related Commonality	C9
Related Parameter	P19

Item Number	V20

Description	The response time required for user interaction with the system varies.
Related Commonality	C10
Related Parameter	P20

Item Number	V21

Description	The operating systems on which a LBS can run.
Related Commonality	C11
Related Parameter	P21

Item Number	V22

Description	The ease with which LBS can effectively be run varies.
Related Commonality	C12
Related Parameter	P22

4.4 Parameters of Variation

This section specifies the parameters of variation for the variabilities listed in Section 4.3. They are organized into the same five subcategories as employed previously: Mesh Generation, Input, Output, System Constraints, Nonfunctional Requirements.

Each parameter of variation is given a unique identifier of the form P followed by a natural number. The corresponding variability is listed and a hyperlink is provided that allows navigation back to the appropriate item in Section 4.3. The final entry for each parameter of variation is the binding time, which is the time in the software lifecycle when the variability is fixed. The binding time could be during specification, or during building of the system (compile time), or during execution of the system (run time). It is possible to have a mixture of binding times. For instance, a parameter of variation could have a binding time of specification or building to represent that the parameter could be set at specification time, or it could be postponed until the given family member is built. The choice of postponing the decision until the build would be associated with the presence of a domain specific language that would allow postponing decisions on the values of the parameter of variation.

4.4.1 Lattice Boltzmann Method Solvers

Item Number	P1

Corresponding Variability	V1
Range of Parameters	OpenMP, OpenCL, CUDA, or MPI are used if LBS is parallel.
Binding Time	Build Time

Item Number	P2

Corresponding Variability	V1
Range of Parameters	Equilibrium can be approximated up to different orders in incompressible or compressible versions.
Binding Time	Build Time

Item Number	P3

Corresponding Variability	V3
Range of Parameters	Single and multiple array storage patterns can be used.
Binding Time	Build Time

Item Number	P4

Corresponding Variability	V4
Range of Parameters	Numerous coefficients for equilibrium distribution function based on number of velocity directions.
Binding Time	Build Time

Item Number	P5

Corresponding Variability	V5
Range of Parameters	LBS models can have up to 3 dimensions.
Binding Time	Build Time or Run Time

Item Number	P6

Corresponding Variability	V6
Range of Parameters	One dimensional models include options of 2, 3, and 5 velocity directions. Two dimensional models include options of 9, 13, and 15 velocity directions. Three dimensional models include options of 15, 19, and 27 velocity directions.
Binding Time	Build Time or Run Time

Item Number	P7

Corresponding Variability	V7
Range of Parameters	SRT, TRT, MRT, BGK collision operators.
Binding Time	Build Time

Item Number	P8

Corresponding Variability	V8
Range of Parameters	Collision and collision free transport equations.
Binding Time	Build Time

Item Number	P9

Corresponding Variability	V9
Range of Parameters	LBS can model one or more fluids.
Binding Time	Build Time

Item Number	P10

Corresponding Variability	V10
Range of Parameters	LBS fluid parameters include Reynolds Number, density, viscosity, time, pressure, force, direction, relaxation rate, turbulence.
Binding Time	Build Time

Item Number	P11

Corresponding Variability	V11
Range of Parameters	Lattice boundary can have reflective or non-reflective conditions.
Binding Time	Build Time

4.4.2 Input

Item Number	P12

Corresponding Variability	V12
Range of Parameters	Input can be graphical, text or file.
Binding Time	Build Time

4.4.3 Output

Item Number	P13

Corresponding Variability	V13
Range of Parameters	LBS can provide 1D, 2D, and 3D rendering of the model.
Binding Time	Build Time or Run Time

Item Number	P14

Corresponding Variability	V14
Range of Parameters	LBS prediction information is output in either text or binary format.
Binding Time	Build Time

4.4.4 System Constraints

Item Number	P15

Corresponding Variability	V15
Range of Parameters	The LBS model can be calculated on the CPU or GPU.
Binding Time	Build Time

Item Number	P16

Corresponding Variability	V16
Range of Parameters	LBS can be run on Windows, MacOS, or Linux versions.
Binding Time	Build Time

Item Number	P17

Corresponding Variability	V17
Range of Parameters	The amount of memory and storage varies between LBS. LBS sizes begin in the low gigabytes.
Binding Time	Build Time

4.4.5 Nonfunctional Requirements

Item Number	P18

Corresponding Variability	V18
Range of Parameters	Each input and each output will have a specified the precision with which the real number is stored and/or calculated.
Binding Time	Build Time or Run Time

Item Number	P19

Corresponding Variability	V19
Range of Parameters	The tolerance allowed for each output produced by the system so that the system produces usable results.
Binding Time	Build Time or Run Time

Item Number	P20

Corresponding Variability	V20
Range of Parameters	The maximum amount of time that the user is expected to wait for the system to produce a result ranges between systems and models from near instantaneous to several minutes or longer.
Binding Time	Specification

Item Number	P21

Corresponding Variability	V21
Range of Parameters	The ease with which LBS can run on Windows, MacOS, and Linux system varies. Some LBS can run on all three, while some can run on multiple or only one operating system.
Binding Time	Specification

Item Number	P22

Corresponding Variability	V22
Range of Parameters	Some LBS require programming skills and domain expert knowledge while some LBS can be run without such skills.
Binding Time	Specification

5 Methodology - DONE SECTION 1st DRAFT

In this project we set out to answer the research questions found in Section 9.1 for the Lattice Boltzmann scientific computing software domain. The process involved systematically measuring and analyzing members of this software family along the quality attributes found in Section 5.2. This included gathering quantitative and qualitative measurements. A goal of the project was to also produce a hierarchical quality assessment for LBM software.

We collected quantitative data using the measures found in the measurement template of Section 3. Some of this was empirical software engineering related data, such as the number of files, number of lines of code, percentage of issues that are closed. Most was gathered by manually observing the software, it's source code, and it's artifacts, while some was gathered using the tools listed in Section 5.5.1.

We also collected qualitative data by interviewing the software package developers and asking them the questions found in Section 9.5. Furthermore, we solicited the assistance of domain experts to better access each software package by leveraging their experience to access the functional and non-functional domain software requirements.

This section begins with the steps of the overall process we used to select, measure and compare LBM software. This is followed by quality definitions and how they were applied in our project. The rest of the section provides an overview of how candidate software packages were selected and filtered, the empirical measurements and software tools, and the analytic hierarchy process (AHP).

5.1 Process

An overview of the assessment process is given in the following steps:

1. Identify a broad list of candidate software packages in the domain. This is discussed in Section 5.3.

2. Filter the software package list. This is discussed in Section 5.4.

3. Gather the source code and documentation for each software package.

4. Collect empirical measures. This is discussed in Section 5.5.

5. Measure using the measurement template. This is discussed in Section 5.5. The measurement template can be found in Section 3.

6. Survey the developers. The developers of each software package of the filtered software list were contacted for voluntary interviews. Four interviews were conducted in total. The interview questions can be found in Section 9.5.

7. Use AHP to rank the software packages. This is discussed in Section 5.6.

8. Analyze the results and answer the research questions. The answers can be found in section 7.

These steps are further detailed in the Methodology for Assessing the State of the Practice for Domain X document found here.

5.2 Software Qualities

Software quality attributes facilitate the measurement and comparison of software packages in this state of the practice assessment. In this report we adopt software quality definitions from various researchers and subject matter expert entities. The quality measurement results are found in Section 6 of this report. Section 7.5 lists recommendations to address software qualities in LBM software packages. The following are the software quality definitions used in this state of the practice exercise, along with comments regarding their quantitative and qualitative measurement.

5.2.1 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 this case effort includes the time spent finding and understanding the installation instructions, the man-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 can also have a positive effect on the measure of installability. Similarly, the ease of uninstallation has an affect on the measure of installability.

5.2.2 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. Software is less likely to have a formal specification if it is not developed by seasoned or professional software developers. Since some software does not have a specification available, the correctness of software cannot always be verified. Despite an absent specification, the correctness of the output of scientific computing software can sometimes be manually verified by applying domain knowledge. A good measure of correctness correlates with the availability of a requirements specification and reference to domain theory, as well as the explicit use of tools or techniques for building confidence of correctness, such as documentation generators and software analysis tools.

5.2.3 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). Simi-

larly to correctness, verifiability is positively correlated with the availability of a specification and with reference to domain knowledge. A good measure of verifiability is further positively 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.

5.2.4 Reliability

Reliability is measured by the probability of failure-free operation of a computer program in a specified environment for a specified time, i.e. 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 the measure of reliability. In turn the presence of descriptive error messages and logs increases the ability to recover the system.

5.2.5 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 like 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.

5.2.6 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 quantitatively measured. Instead the documentation of each software package was scanned for information that alludes to a consideration of performance, such as parallelization tools.

5.2.7 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). A good measure of usability correlates with the presence of documentation, specifically tutorials, manuals, and defined user characteristics, and user support. Preferably the user support model has avenues to contact developers and report issues.

5.2.8 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 this state of the practice analysis maintainability is measured by the quality of documentation artifacts, and the presence of version control and issue tracking. These qualities 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.

5.2.9 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). This state of the practice assessment did not quantitatively measure modifiability. 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 asked if any measures had been taken.

5.2.10 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, smaller components with well defined interfaces, is important. For this state of the practice assessment, a good measure of reusablity correlates with an increased number of code files, and the availability of API documentation.

5.2.11 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, 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.

5.2.12 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 some form of specification to be available. The quality refers to keeping track of information as it changes forms or relates between artifacts. This state of the practice assessment did not quantitatively measure traceability. Developers were asked in interviews how documentation fits into their development process.

5.2.13 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, the presence of development process and environment documentation, and software package version release notes.

5.2.14 Reproducibility

Software achieves reproducibility if another developer can take the requirements documentation and re-obtain the same software artifacts (Benureau and Rougier, 2017). In this context, reproducibility refers to the similarity between artifacts. Reproducibility can also refer to the output of the software, irrespective of the artifacts, where the scientific results are compared between software implementations, or between software and manual implementations. This state of the practice assessment did not quantitatively measure reproducibility. Developers were asked in interviews if they have any concern that their computational results wont be reproducible in the future, and if they have taken any steps to ensure reproducibility.

5.2.15 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. Developers were asked in interviews 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 ot achieve it.

5.3 Identify Candidate Software

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.

When forming the list and reviewing the candidate software the following properties were considered:

1. The software functionality must fall within the identified domain.
2. The source code must be viewable.
3. The empirical measures listed in Section 8 should ideally 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 46 packages, including a few packages that were later found to not have publicly available source code, or to be incomplete.

5.4 Filter the Software List

In order to reduce the number of members of the candidate software list, 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) are 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.

Name	Released	Updated
DL_MESO (LBE)	unclear	2020 Mar
waLBerla	2008 Aug	2020 Jul
Sailfish	2012 Nov	2019 Jun
Palabos	unclear	2020 Jul
OpenLB	2007 Jul	2019 Oct
MechSys	2008 Jun	2020 Jul
LUMA	2016 Nov	2020 Feb
Ludwig	2018 Aug	2020 Jul
lettuce	2019 May	2020 Jul
pyLBM	2015 Jun	2020 Jun
lbmpy	unknown	2020 Jun
TCLB	2013 Jun	2020 Apr
ESPResSo	2010 Nov	2020 Jun
ESPResSo++	2011 Feb	2020 Apr

Table 1: Alive Software Packages

While the initial list had 46 packages, filtering by scope, usage, and age, decreased the size of the list to 23 packages. Many of the 23 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, and the project was out of scope or not up to a standard that would support incorporating them into this study. Of the remaining 23 packages that were studied, some were kept on the list despite being marked as dead due to their prevalence on authoritative lists on LBM software and their surface excellence, specifically the clearly considerable time that was put into these projects. The final list of software that was 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.

Name	Released	Updated
SunlightLB	2005 Sep	2012 Nov
MP-LABS	2008 Jun	2014 Oct
LIMBES	2010 Nov	2014 Dec
LB3D	unclear	2012 Mar
LB3D-Prime	2005	2011 Oct
LB2D-Prime	2005	2012 Apr
LatBo.jl	2014 Aug	2017 Feb
HemeLB	2007 Jun	2018 Aug
laboetie	2014 Nov	2018 Aug

Table 2: Dead Software Packages

There is considerable variation among these software packages, including their intended purpose, size, user interfaces, and their base programming languages. For example, the OpenLB software package is predominantly a C++ package that makes use of hybrid parallelization and was designed to address a range of computational fluid dynamics 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 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).

5.5 Empirical Measures

The quality measurements in this assessment rely on the gathering and analyzing of raw and processed empirical data related to the research questions listed in Section 9.1.

All of the quality measurements that are part of the AHP analysis are in this category, while qualitative data gathered during interviews with developers is not. This part of the assessment focuses on data that is reasonably easy to collect. Much of the data was gathered by manually reviewing the artifacts of each software package, while some was gathered using freeware tools listed in Section 5.5.1. The data that was gathered is listed in the measurement template found in Section 3. Some of the data required processing other data within the template, including the status of the software package which relied on the last commit date, the percentage of issues that are closed which relied on the number of open and closed issues, and the percentage of code that is comments which relied on the number of total lines and comment lines in text-based files. The complete measurement template data was then analyzed using the grading template found in Section 4, the output of which was analyzed using the AHP described in Section 5.6.

5.5.1 Software Tools

Most of the measurement template data was gathered by observing GitHub repository metrics and software package artifacts, or by processing data gathered using freeware tools. Data in the final three sets of the measurement template was collected using these tools. 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. Details on installing and running these tools can be found in the Guide to Empirical Measures file in the AIMSS repository.

5.6 Analytical Hierarchy Process

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 found in Section 3 using the grading template found in Section 4. AHP performs a pairwise analysis using a matrix and generates an overall score as well as quality scores for each software package. Smith et al. (2016) 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. The README file of the tool, which includes requirements and usage information, is found here.

6 Results - DONE SECTION 1st DRAFT

This section presents the AHP results for qualities that were quantitatively measured. Qualitiative data from interviews with some developers is also found in this section. Working quality definitions are found in section 5.2.

6.1 Installability

All of the software packages that were tested have installation instructions. As noted previously, many of the 23 software packages removed from the original list of 46 software packages are missing documentation, including installation instructions.

All 23 packages can be installed on some Unix-like systems. Seven packages could be installed on Windows, and 5 on macOS. Operating system compatibility is found in the documentation of 19 software packages. For this state of the practice assessment, all but one of the software packages were tested on Ubuntu. The remaining package was tested on CentOS.

Of the software packages that were tested, most installation instructions are located in one place, often in an instruction manual or on a web page. Sometimes incomplete installation instructions are found on a home page, with more detailed instructions located on another web page or within formal documentation. Maintainability and correctness of these instructions could be improved if they were limited to one location.

All but two of the software packages have an automation procedure for at least part of the installation process. Most of these packages use make to automate the installation. A couple of packages use custom scripts for automation.

Errors encountered during the installation process were sometimes quickly fixed thanks to descriptive error messages. At other times, error messages were vague and difficult to troubleshoot. Fourteen software packages definitively broke during installation. A couple

of packages 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 they were available. The installation of most software packages could be recovered if a descriptive error message was displayed. Only three software packages 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.

About half of the installation instructions are not written as if the person doing the installation has none of the dependent packages installed, and many software packages do not list all of their dependencies. Many dependencies needed to be manually installed. Sometimes only an error message during the installation process informs the user of the requirement of these packages. A detailed rewrite of the installation instructions from the point of view of installation on a clean operating system is suggested.

Sixteen software packages require less than 10 dependencies to be installed. All but one software package require less than 20 dependencies. Some packages may automatically install additional dependencies in the background. Eighteen 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 the dependency versions are specified. Fifteen software packages do not have detailed instructions for installing dependencies.

Sixteen software packages have less than ten manual installation steps. If commands for the installation of dependencies is streamlined then none of the software packages take more than 20 steps to install. The average number of steps is about 8, and the fewest is 3.

All but six of the software packages have a way to verify the installation. Twelve have some sort of tutorial examples that can be run by the user. Most of these generate output that can be compared to the documentation. Some packages do not have a way to verify the output of the tutorials, and the user is left to assume correctness if there is no error

prompted during the tutorial. Some other ways of installation validation include validation scripts, automatic validation after the installation, and instructions to manually review the file system.

Instructions for uninstallation is easily found for only one of the software packages.

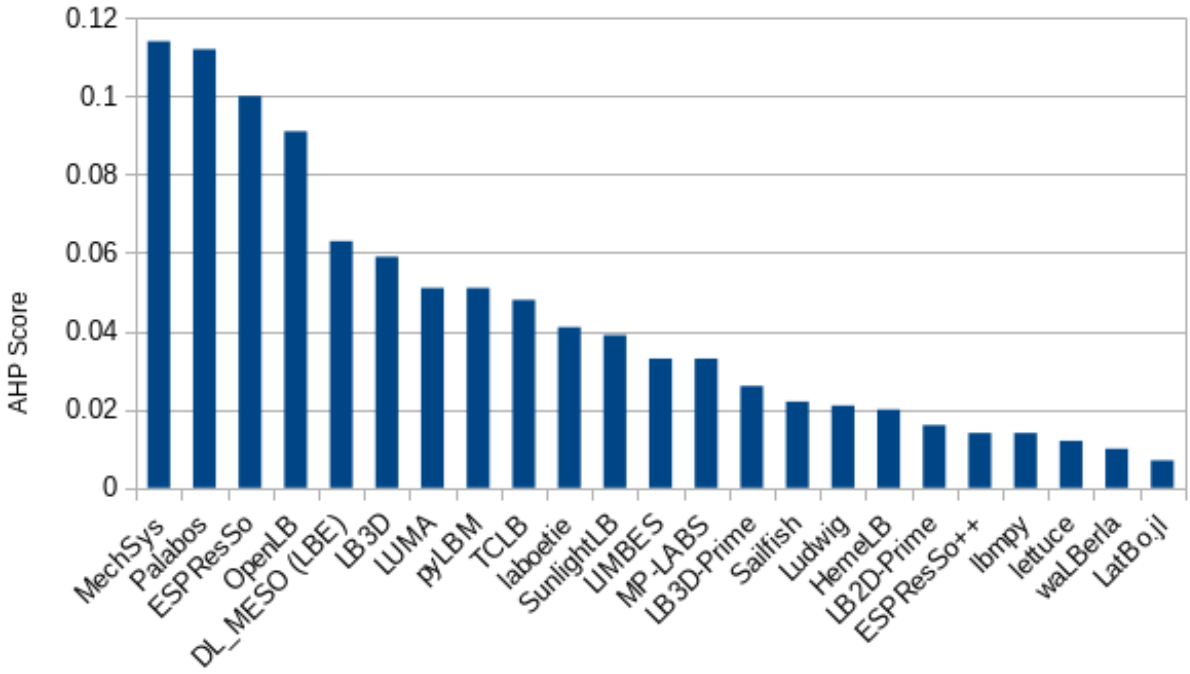

Figure 2: AHP Installability Score

Figure 2 shows the installability ranking of the software packages using AHP. Software packages with a higher score tend to have one set of linear installation instructions that is written as if the person doing the installation has none of the dependencies installed. The instructions often have compatible operating system versions listed and include instructions for the installation of dependencies. The top ranked packages also often incorporate some sort of the automation in the installation process and have fewer manual installation steps. The number of dependencies a package has does not correlate with a higher score. Installation validation, often through test examples with expected output, is correlated with a higher score.

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. Errors that may be encountered during the installation process need to prompt detailed messages or logs. Once a software package is installed, either a transparent automatic validation needs to be performed or the user needs to be able to perform a manual validation using test examples and output. Finally, uninstallation instructions should be included in the documentation.

6.2 Surface Correctness and Verifiability

Sixteen of the software packages have a requirement specification or explicitly reference domain theory. This information is often found in a user manual, on a web page, or is mentioned in publications by the developers. In the latter case the user may need to spend significant time to find this information.

Interviews with developers confirmed that many of the software packages were developed by domain experts with backgrounds in physics, mathematics and mechanical engineering. Many did not have formal software engineering education. Some development teams include computer scientists. Despite a lack of visible domain documentation and a resulting lower surface correctness and verifiability score, many software packages do have a strong domain background.

Document generation tools are explicitly used by 12 software packages. Sphinx is used by 8 of them, and Doxygen is used by 7. Some of the packages use both.

Tutorials are available for all but 5 of the software packages. Generally they are linearly written and easy to follow. However, only 8 tutorials provided an expected output and it is not possible to verify the correctness of the output of the software packages that are missing

this key information. In this case the user cannot be sure of correctness and may need to assume it if there are no immediately visible errors.

Unit tests are only explicitly available for one of the software packages. Code modularization of most packages allows for a user 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 documentation or presence of continuous integration tools and techniques alludes to a more refined development process where faults are isolated and better recognized. On the surface only 2 of the packages mentioned applying the practice of continuous integration in their development process.

Interviews with the developers of software packages suggest a more frequent use of both unit testing and continuous integration in the development processes than initially surveyed. For example, OpenLB, pyLBM, and TCLB use such methods during development despite it not being explicitly clear from a surface analysis.

Figure 3 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 in their documentation, and most provide expected output for these tutorials. Only the top ranked package, Ludwig, provided unit testing. It and the second ranked package, ESPResSo, explicitly incorporated continuous integration in the development process. During developer interviews it was discovered that some other packages also use continuous integration and unit testing, despite not being explicitly stated in documentation. Such packages, including OpenLB, pyLBM, and TCLB, have higher true correctness and verificablity than is noted from surface analysis.

The inclusion of requirements specification and theory documentation greatly benefits

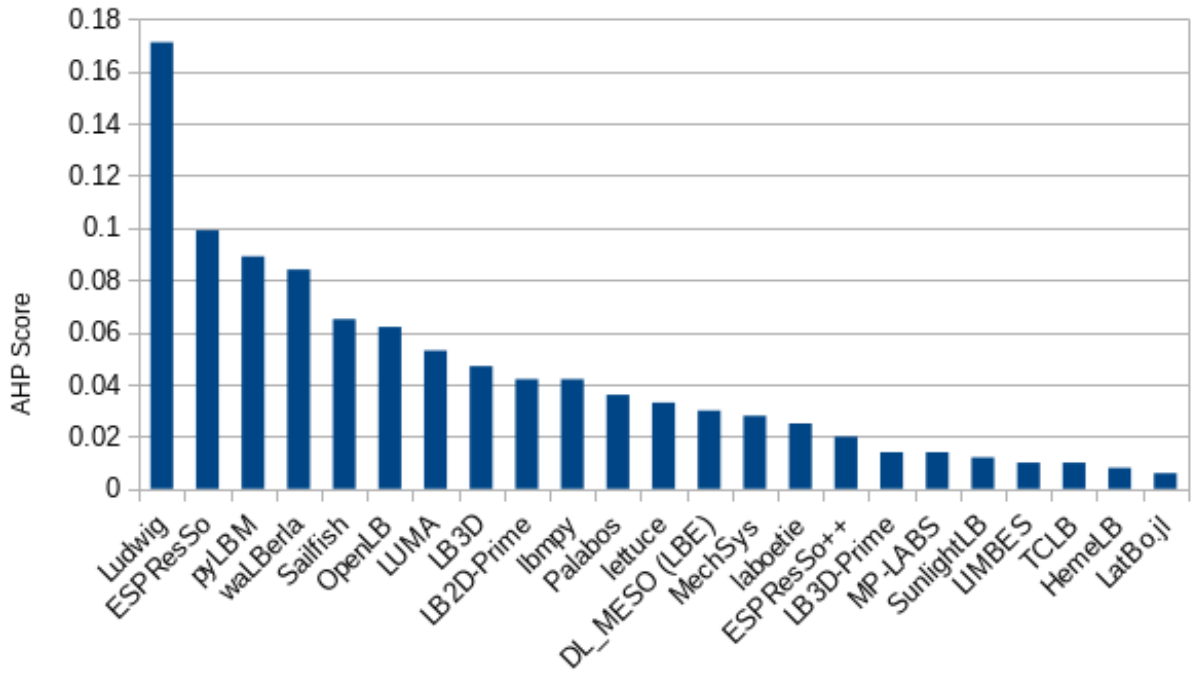

Figure 3: AHP Surface Correctness and Verifiability Score

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, along with expected output, further helps users verify the software and have confidence in its correctness. Unit testing documentation and capability, as well as the use of continuous integration tools and techniques such as Bamboo, Jenkins, and Travis CI, help verify correctness. Several interviewed developers alluded to difficulty with testing large numbers of features, and some even manually tested output. The use of well defined unit testing tools could decrease the time spent testing some feature.

6.3 Surface Reliability

Errors occurred when installing 16 of the software packages, and every instance prompted a message. Error messages included unrecognized commands (even when following the in-

stallation guide), missing links, missing dependencies, syntax errors in code files, and vague output. Several automatic installation processes could not find and load dependency packages. The installation files pointed to outdated external repositories. Seven of the installations were clearly recovered and one of the installations was assumed to be recovered due to the absence of installation verification. The installation of eight of the software packages could not be recovered. Most of these installations could not reach a dependency repository, had an incompatible error in their core installation files, or prompted vague error messages.

Of the 13 software packages that installed correctly and had tutorials, 4 broke during tutorial testing. All four error resulted in an error message being displayed. One error was due to a missing tutorial dependency, another was due to an invalid command despite following the tutorial, and the final two errors were execution errors. Of the 4 broken tutorial instances, only the one that was missing a dependency was recoverable.

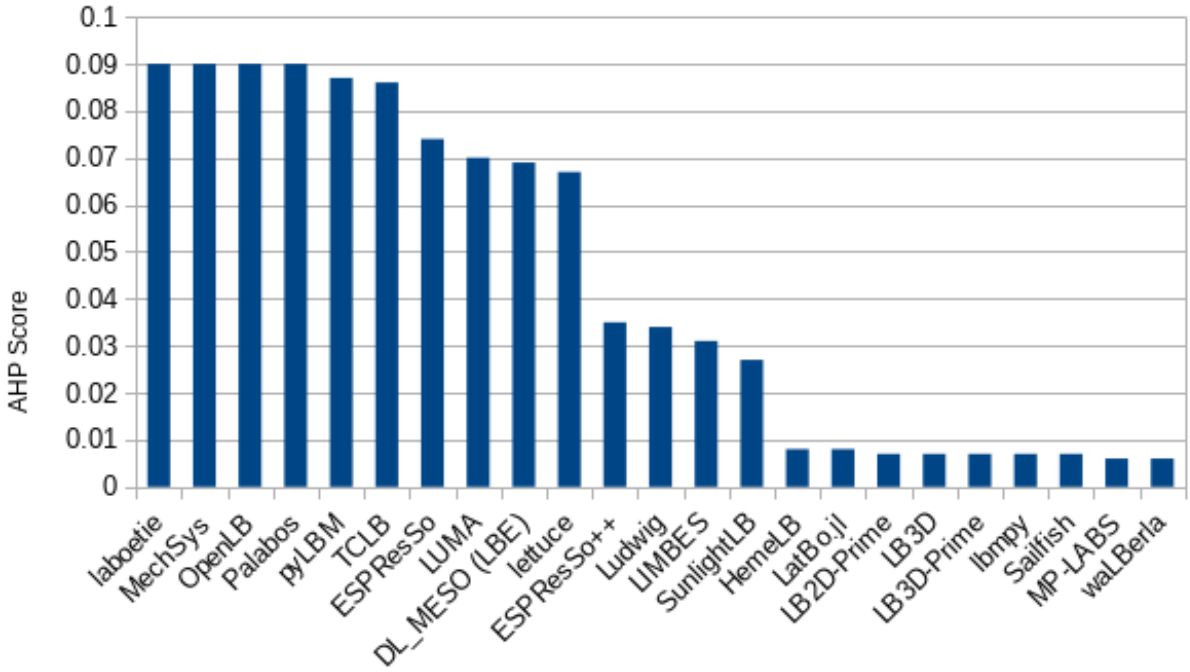

Figure 4: AHP Surface Reliability Score

Figure 4 shows the surface reliability ranking of the software packages using AHP. Soft-

ware packages with a high score did not break during installation, or the installation was recoverable. All of the top 5 ranked packages had tutorials. One of these packages, pyLBM, broke during tutorial testing, but a descriptive error message helped in recovery.

Overall, lower ranked software packages were lacking clear documentation, testing or tutorial examples, and descriptive error messages, and had broken automatic dependencies. Thus, regarding surface reliability, software packages would benefit from clear up-to-date installation and tutorial documentation, and descriptive error messages.

6.4 Surface Robustness

The software packages were tested to handle 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.

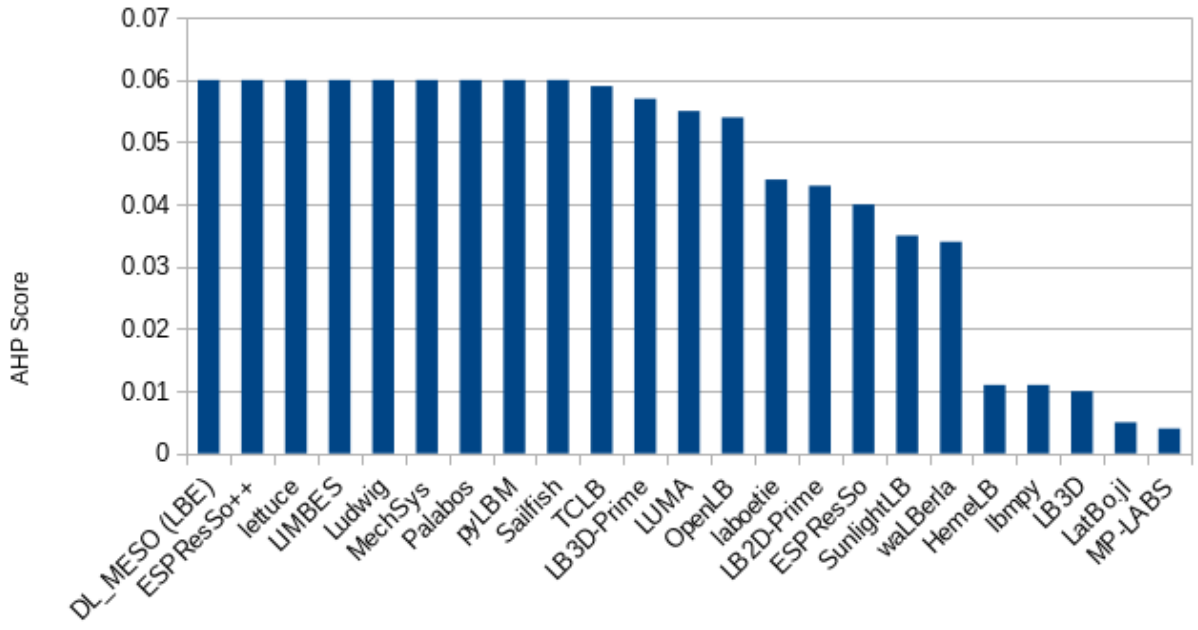

Figure 5: AHP Surface Robustness Score

Figure 5 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 had not installed correctly, so their score here may not be a true reflection of runtime robustness. Similarly, all software packages that installed correctly and require plain text input files handled an unexpected change, a replacement of new lines with carriage returns, to these input files correctly.

6.5 Surface Performance

Each software package was not quantitatively measured for surface performance. Although the software packages all apply Lattice Boltzmann methods to solve scientific computing problems, the packages focus on varied domains, with varying parameters, and are technically different from each other. Due to this, a comparison of performance is not appropriate. Instead, in this project we looked through each software package's artifacts for evidence that performance was considered. The artifacts of 17 software packages mentioned the use of some sort of parallelization for execution. This included GPU processing and the CUDA parallel computing platform. 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 some software packages required CUDA and GPU processing, some had 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.

6.6 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 18 software packages have a tutorial and 13 have a user manual. The tutorials vary in scope and substance, and 8 include an expected output to compare user results to. Most user manuals are in the form of a file that can be downloaded, while some are rendered on a web-page. Some packages do not have a user manual but do have useful documentation distributed on their web-pages. Expected user characteristics are documented in 4 software packages. Users are typically scientists or engineers trying to test a new application. Their background is often physics, chemistry, biophysics, or mathematics. All but one of the packages have a user support model, and several of them have multiple avenues of user support. The most popular avenue of support is git, followed by email and forums. One software package has a FAQ page.

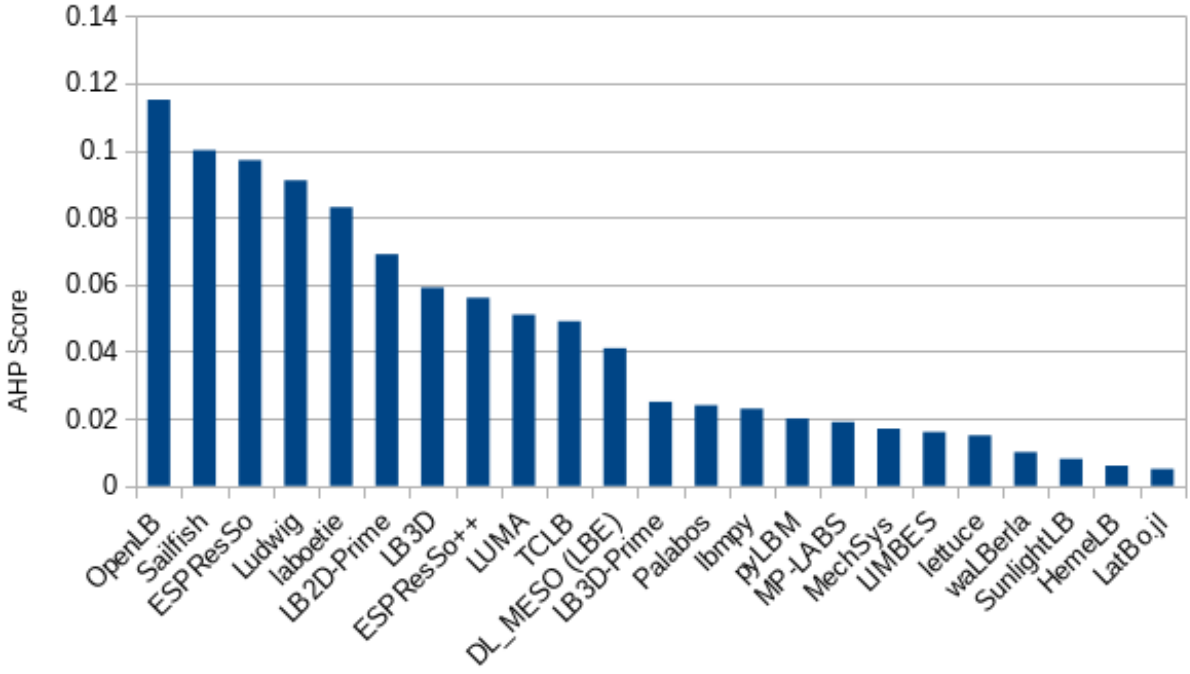

Figure 6: AHP Surface Usability Score

Figure 6 shows the surface usability ranking of the software packages using AHP. Software packages with a high score have a tutorial and user manual, and sometimes have documented user characteristics, and have at least one user support model, with many having several.

Interviews with developers revealed several usability issues. Some users have misunderstood boundaries of LBM and computational fluid dynamics in general, and have combined or applied methods that are not physically sound together. Sometimes users have applied LBM to poorly defined or inappropriate fluid dynamics problems. The users do not realize the limitations of the methods, of the software, and of how much time and effort is required to properly model a problem that can technically be modeled by the software. The software is not as easy as out of the box plug and play. Developers of some of the software packages mitigated this by editing the source code to prevent unsound combination from being applied, 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.

Some additional infrequent software usability issues were commented on. Users have had trouble with installation and understanding how to maneuver the interfaces and how to set up or 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 requirement specifications.

One software package changed its scripting language to Python to make it more usable. The developer commented that this was the biggest step in addressing usability, commenting that Python was a much easier language for users to learn and understand.

6.7 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 version

numbers. This information could be used to better troubleshoot software issues and organize documentation. All but 3 software packages have source code release and documentation version numbers.

Information on how code is reviewed, or how to contribute to the project was also noted. In total, eleven 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 22 software packages, 15 of which use git, 6 use email, and one uses SourceForge. Most software packages that use git have most of their issues closed, and only 3 have less than 50 percent of their issues closed. Furthermore, most packages that use git for issue tracking use GitHub as a version control system, one package uses CVS for issue tracking, and 8 packages do not appear to use any such system.

Software package code files were further measured for the percentage of code that is comments. Packages with a higher percentage of comments were designated as more maintainable. Comments represent more than 10 percent of code files in 15 packages, and the average percentage of comments is 13.7.

Figure 7 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, and use an issue tracking tool and version control system. These packages also appear to reasonably handle tracked issues, having most of their issues closed. Their code files are well commented with more than 10 percent of the code being comments.

Interviews with developers revealed that most projects do have a defined process for accepting contributions from current team members or those that at some time had been linked to the packages or developers. The packages rarely get contributions from outside developers, but the process would be similar as for the aforementioned group.

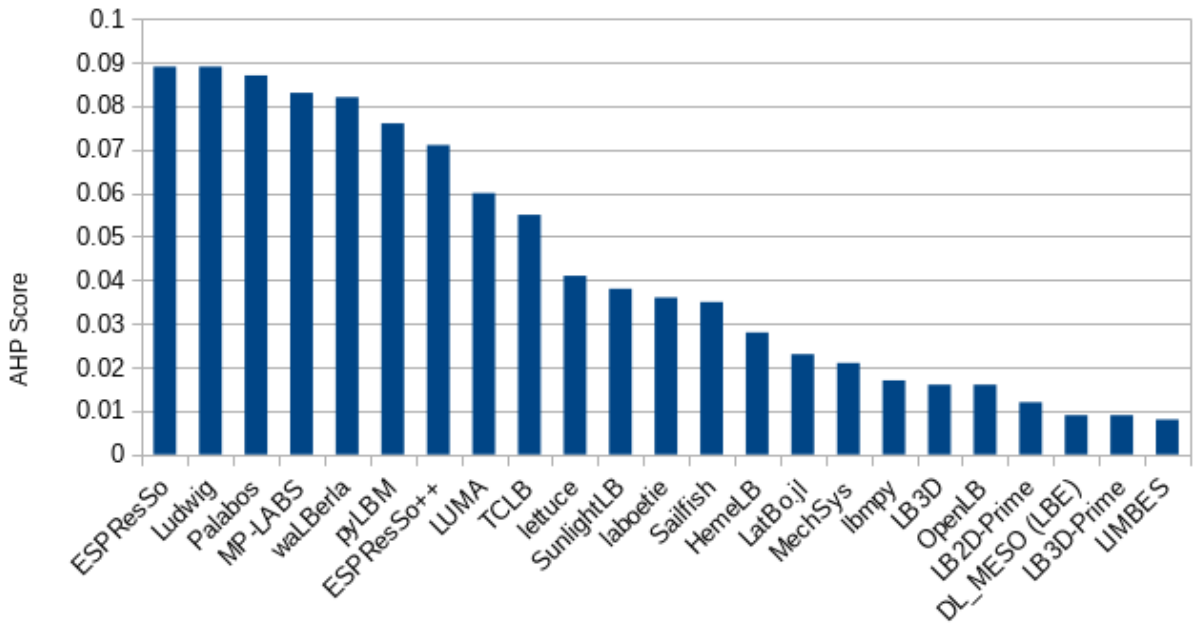

Figure 7: AHP Maintainability Score

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.

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 very happy to see spontaneous contribution from outside of their organization, but now it is currently not the case.

Furthermore, maintainability has been addressed by increasing source code modularity, reducing duplicate information, and improving abstraction. Several software packages have had large sections of their code base redeveloped with languages that the developers felt are more understandable and readable, and that are better supported and have preferred dependencies. Data structures have also been redeveloped and storage has been improved. One software developer mentioned that the mathematical foundations that the geometries

and models are designed from have been re-evaluated and optimized.

6.8 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.

A high degree of code modularity and abstraction was noted by one developer as a measure to ensure ease of future changes. They also noted that some of the code base was transitioned from c to c++, which could ease modifiability of that software package. Another developer noted that their software package was designed to easily allow for the addition of some LBM features, but changes to major aspects of the LBM models would be difficult. 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 in order to stimulate the development of third-user extensions. The Palabos package identifies as as a development framework for modeling problems in various computational fluid dynamics areas.

6.9 Reusability

The total number of source code files was measured for each software package. A larger number of source files was associated with increased reusability since this generally represented either more software package functionality or better modularization. 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.

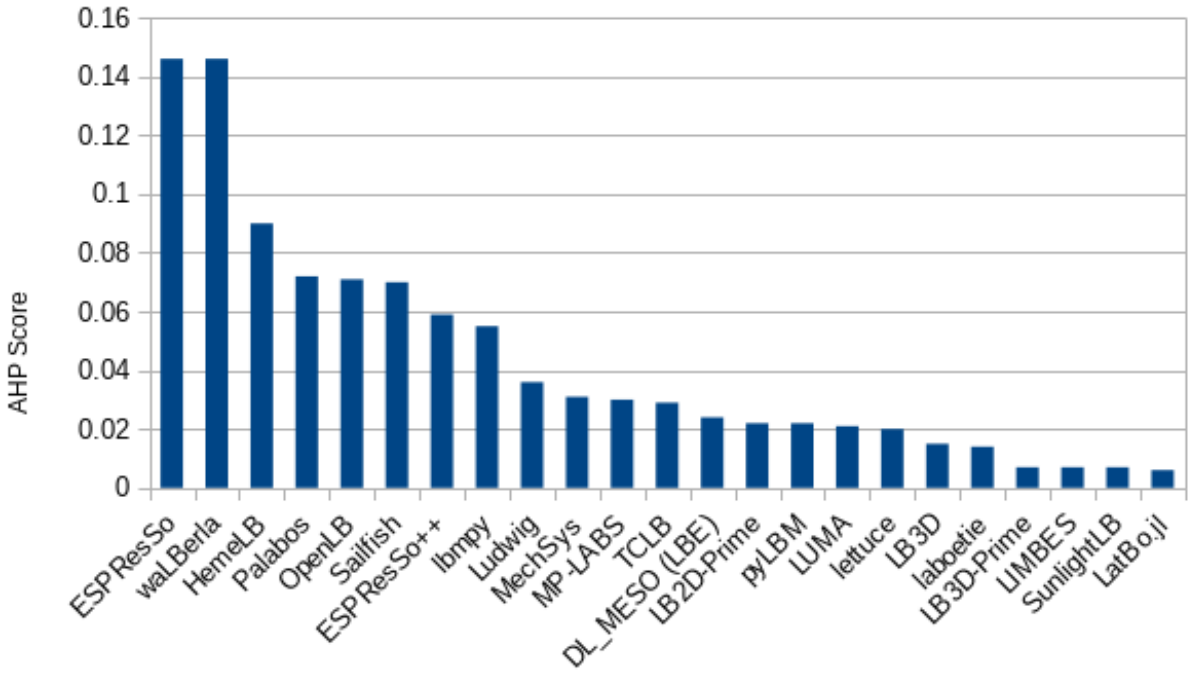

Figure 8: AHP Reusability Score

Figure 8 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 as well as modeling that does not use LBM. For this reason a comparison between other software packages is not on a level field, but regardless of this these packages do have an abundance of reusable components.

There was a strong focus on modularity when designing the waLBerla framework in order to enhance productivity, reusability, and maintainability (Bauer et al., 2021). The software design has enabled for waLBerla to be successfully applied in several projects as a basis for various extensions (Bauer et al., 2021).

6.10 Surface Understandability

Ten random source code files of each software package were reviewed for several measures. This measure of surface understandability may not perfectly reflect each package due to this limitation. All of the packages appear to have a consistent indentation and formatting style. Only LUMA and HemeLB explicitly identify coding standards that are used during development. Generally, the software packages use consistent, distinctive, and meaningful code identifiers. Only 4 packages appear to use vague identifiers. Hard coded constants were observed in the source code of 12 packages. The constants are used for various parameters, mathematical constants, and matrix definitions. All of the packages are well commented. Comments clearly indicate what is being done. Domain algorithms are noted in the source code of 11 packages. All of the packages were well modularized. When observing the source code files, it was found that 13 of the packages had a consistent style and order for function parameters.

Figure 9 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 hard coded constants, and explicitly identify mathematical and LBM algorithms. Their comments are clear and indicative of what is being done. The source code is well modularized and structured.

Software developers noted that in their opinion users have generally found their packages to be understandable. One developer 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 update the documentation to prevent misunderstanding the methods. Similarly, another developer noted that some users had issues setting up parameters for LBM schemes. The solution was to this was to update the interface where these parameters are set, as well as adding functionality to test the stability of the parameters. A third developer noted that some users lack the background knowledge to easily model

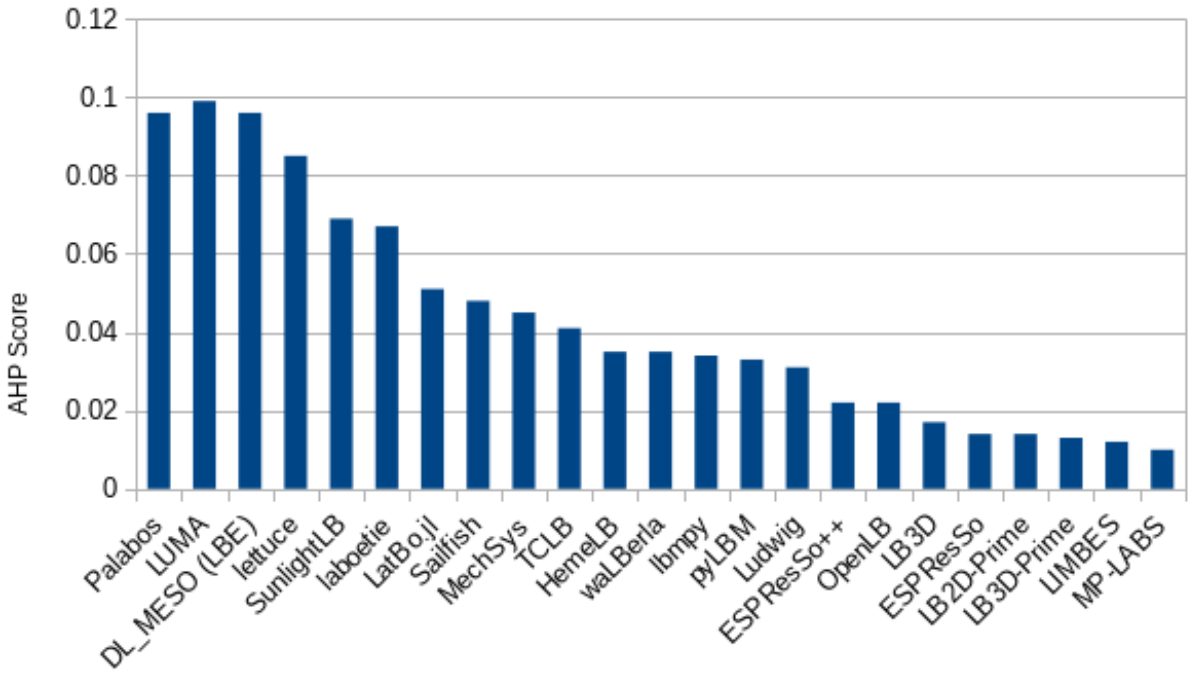

Figure 9: AHP Understandability Score

flow problems using their software. A frequently asked questions section was added to the source package website to help users find answers to common questions. The package also has detailed documentation, including guides and usage requirements specification.

6.11 Traceability

Software packages were not quantitatively measured for traceability. In this project we asked developers to comment on traceability when we interviewed them. They were asked to comment on their documentation and how it fits into their development process.

One developer noted that all major additions to their package had accompanying changes to artifacts, especially noting user documentation, and how it is updated to note major usability issues. Considerable effort had been put into the documentation. They want to lower the entry barrier for new developers so there is considerable documentation aimed at

developers, and it is frequently updated. This documentation informs developers on how to get started, 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.

Another developer noted the importance of documentation for both user and developers of their software. New features are always well documented in the artifacts. The developers use documentation to stay up to date on the status of the software package, and to expand or use features, like computational models or algorithms. This is necessary so that the coding standard for these models is kept consistent as developers come and go.

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 had negative affect on the documentation. Most of these developers are scientific researchers evaluated on the scientific papers that they produce. Writing and updating documentation is something that is done in free time, if that time arises, and if writing documentation is a concern. 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.

One developer noted the use of documentation generators like Doxygen. It would be advisable for more projects to use such automatic document generation tools, if they are a good technological fit.

6.12 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 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 9 of the software packages.

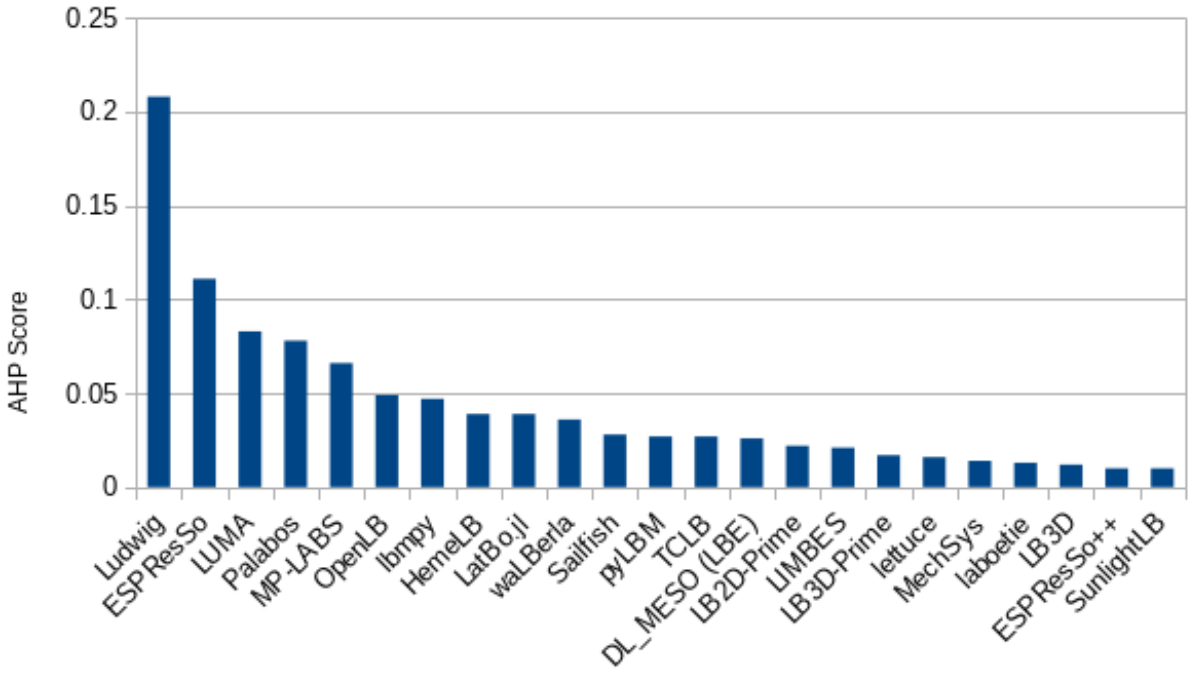

Figure 10: AHP Visibility and Transparency Score

Figure 10 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.

Developers were asked to comment on the obstacles in their development process. One developer 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 would help with code modifiability and reusability. Updating the development process would help resolve this issues and prevent such issues in the future. Improving the visibility of software changes and the peer review process would also help. Improving the software engineering education or experience of developers is also an idea that was brought up by several developers. Specifically, developers should always write code that is decoupled and modular, and should keep in mind the visibility of their contributions by better updating documentation and ensuring that their contributions are transparent to the rest of the developers. This would help catch issues in the code contributions, and improve source code maintenance.

According to one developer, some obstacles to the development processes of their package had been overcome by the introduction of continuous integration practices, and a peer review process for contributions. These practices decrease development and maintenance times. Another developer 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, the use of which is now part of the development process.

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.

None of the software packages whose developers were interviewed have a formal software development model. The packages all have fairly small development teams. These teams do accept outside contributors, but generally the teams are tight-knit, often working at the same institution, although one of the packages has an international team. One developer

noted that while no formal model is used, their development model is something similar to a combination of agile and waterfall development models.

The developers noted similar project management processes. In teams of only a couple of developers, the addition of new features or major changes were discussed with the entire team. Projects with more than a couple of developers had lead developer roles. These lead developers would review potential additions to the software. The software packages use GitHub for managing the project. Typically there are several development branches as well as the master branch.

6.13 Reproducibility

Software packages were not quantitatively measured for reproducibility. In this project we asked developers to comment on reproducibility when we interviewed them.

Developers were asked if they have any concern that their computational results would not be reproducible in the future, and if they had taken any steps to ensure reproducibility.

One developer noted that they compare the results of their methods against manually calculated results. These tests comparing these results are automatically run for all source code changes that are suggested. The tests are run when a pull request is opened on GitHub. Even once these tests are complete, a peer review process is done 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 compared daily for correctness, ensuring that the system output reflects the expected output.

Several developers noted that they currently do not have a system in place to test for reproducibility, but it is of interest and could be implemented in the future. Generally, the mathematical foundations of the models are verified, but the output of the software packages is not compared to other output. Depending on the package and how it outputs LBM solutions, it 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.

The source code and artifacts of some software packages may be reproducible. There is considerable variance in the quality of the software specification and other developer documentation. Some packages are well detailed, and translating the specification into source code will produce similar results across developers.

6.14 Unambiguity

Software packages were not quantitatively measured for unambiguity. In this project we asked developers to comment on unambiguity when we interviewed them.

Developers were asked 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.

One developer noted that their documentation was meant for users that are already familiar with underlying physics and computational fluid dynamics methods, and that these concepts are not explained in detail. The documentation focuses solely 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 a those with a minimum knowledge of the underlying physics to learn to use the package. If new users have technical questions these can be addressed in further revisions of the documentation. New developers should also find that the code documentation is reasonably clear and useful. Information that is missing, like information explaining dependency packages, is reasonably referenced.

Another developer also noted that their documentation was in reasonable shape, but they would really appreciate more user feedback to improve it. They also noted that they believe a lack of knowledge of the underlying physics and computational fluid dynamics 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.

6.15 Overall Quality

Figure 11 shows the overall ranking of the software packages using AHP. Software packages with an overall high score had ranked high in at least several of the individual qualities that were quantitatively measured.

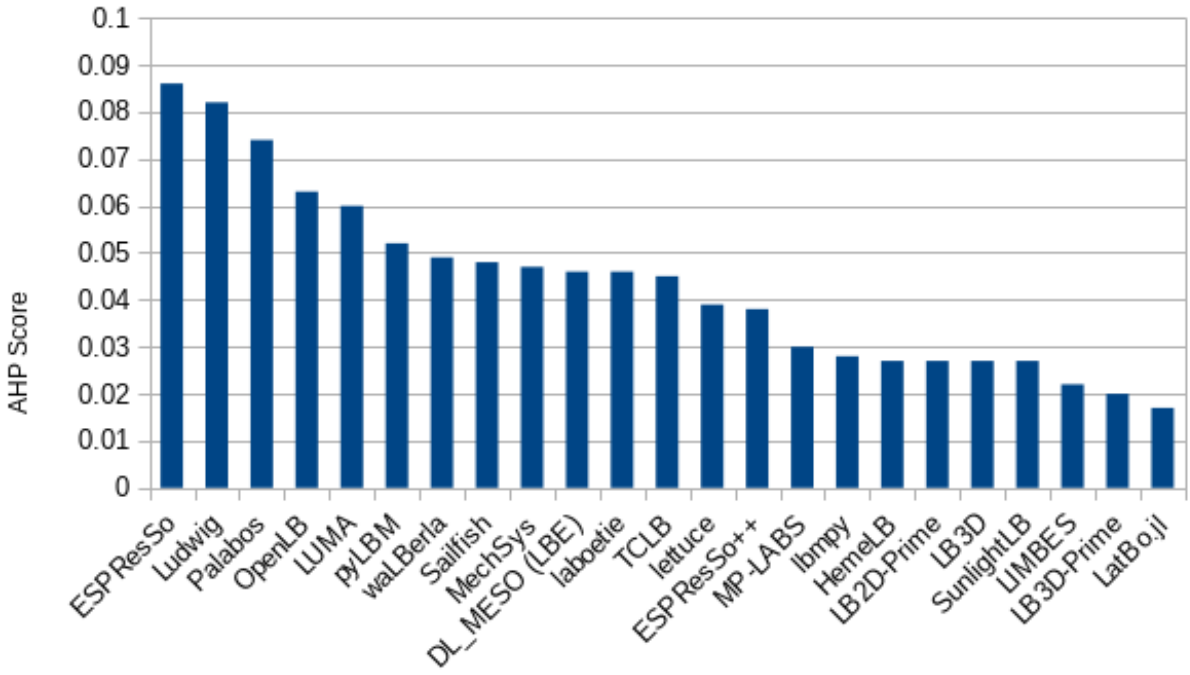

Figure 11: AHP Overall Score

Looking at the top three ranked packages. ESPResSo had achieved a relative high score in installability, surface correctness and verifiability, 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, and understandability.

Section 7.5 further analyzes these findings and offers some software quality recommendations for future development of LBM software.

7 Answers To Research Questions

7.1 Artifacts Present - DONE 1st DRAFT

The software packages were examined for artifacts. Below we have grouped these into common, less common, and rare artifacts. Common artifacts were found in 16 to 23 of the software packages. Less common artifacts were found in 8 to 15 of the software packages. Rare artifacts were found in 1 to 27 of the software packages.

7.1.1 Common Artifacts

The following artifacts were commonly found in the 23 software packages that were tested. All of the top four AHP ranked packages, ESPREsSo, Ludwig, Palabos, and OpenLB have each of these artifacts, except for a clear requirements specification or theory notes. Only three of the packages were found to have that artifact.

- Authors/Developers List
- Bug Tracker
- Dependency Notes
- Installation Guide/Instructions
- License
- List of Related Articles/Publications
- Makefile / Build File
- README File
- Requirements Specification / Theory Notes
- Tutorial

These common artifacts contribute to the quality of the software in various ways. A list of authors and developers helps potential users and contributors more easily contact project members to answer questions affecting many software qualities. A bug tracker helps with organizing improvements to the software, also improving many software qualities. Dependency notes help users install the software, as does an installation guide. Makefiles or other automated build files decrease human error in the installation process. Visible licenses promote

usage of the software. Requirements specifications, linked theory notes, and related articles and publications help clarify the software and make it more understandable, decrease ambiguity, and help to verify its correctness. README files also help with understandability. Meanwhile tutorials help users become familiar with using the software.

7.1.2 Less Common Artifacts

The following artifacts were less commonly found in the 23 software packages that were tested. The top four AHP ranked packages have most of these artifacts. Only one of those four packages does not have a user manual or guide, but it does have a detailed and informative website. Another one of the top four packages does not have any visible design documentation. A third package from that list does not appear to use a version control system. It is probably that such a system is in use but is not visible outside of the development team since the package website notes a current version number.

- Change Log / Release Notes
- Design Documentation
- Functional Specification/Notes
- Performance Information/Notes
- Test Plan/Report/Script/Data/Cases
- User Manual/Guide
- Version Control

These artifacts also contribute to software quality. A change log or release notes improve traceability of the software. Design documentation helps with maintaining, modifying, and reusing the software. Version control also helps to improve these qualities, as well as traceability. Functional specifications and notes clarify the software, contributing to usability and understandability. A user manual also benefits those qualities and with installability, and correctness and verificability. Performance notes suggest that performance was considered when developing the software. Test plans, scripts, and cases, help verify correctness.

7.1.3 Rare Artifacts

The following artifacts were rarely found in the 23 software packages that were tested. It is not common for the top four AHP ranked packages to have many of these artifacts. None of the top four packages have any explicit API documentation. Three of these packages have information on contributing to the project. Two of them have a FAQ section or forums. One has verification and validation notes and a video guide explaining how to use the software.

- API Documentation
- Developer/Contributing Manual/Guide
- FAQ / Forum
- Verification and Validation Plan/Notes
- Video Guide (including YouTube)

Although these artifacts were rarely found in our set of LBM software, they also contribute to software quality. API documentation helps with reusing the software. Developer and contributor manuals and guides help with maintainability, visibility and transparency. FAQs and forums increase usability of software. Verification and validation notes can be used to help check if a system meets specifications. Video guides can contribute to many software qualities, depending on the content of the video.

7.2 Tools Used - DONE 1st Draft

The purpose of tools is to support the development, verification, maintenance, and evolution of software, software processes, and artifacts (Ghezzi et al., 1991). Many tools are used by current LBM software packages. The tools noted here are subdivided into development tools, dependencies, and project management tools.

7.2.1 Development Tools

These 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). The following type of development tools were explicitly noted in the artifacts or web pages of the 23 LBM packages that were assessed. It is likely that other tools, such as debuggers, were used but are not specified here.

- Continuous Integration Tools
- Code Editors
- Development Environment
- Runtime Environments
- Compilers
- Unit Testing Tools
- Correctness Verification Tools

The use of the above tools can verify the correctness of software during its development. Only two of the software packages that were assessed mentioned using 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 explicitly noted the use of unit testing, but did not specify the tool that was used. Likewise the use of a tool for verifying the correctness of output was noted by one of the developers. It is likely that similar tools were used when developing other members of the software package list.

7.2.2 Dependencies

The following types of dependencies were explicitly noted in the artifacts or web pages of the 23 LBM packages that were assessed. It is possible that other types of dependencies are part of these software packages but are not clearly specified in their artifacts or web sites and because of that they are not listed here.

- Build Automation Tools
- Technical Libraries
- Domain Specific Libraries

Most of the software packages use some sort of build automation tools, most commonly Make. They also all use various technical and domain specific libraries. Technical libraries include visualization, data analysis, and message passing libraries among others. Domain specific libraries are scientific computing libraries. Specific libraries are not listed here but they fall into the above categories. Libraries that are not explicitly stated in any artifact may fall outside of the categories.

7.2.3 Project Management Tools

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. The following types of project management tools were explicitly noted in the artifacts, web pages, or interviews with the developers of the 23 LBM packages that were assessed. As with development tools and dependencies, it is possible that other types of project management tools were used to coordinate and manage the projects but are not specified and because of that they are not listed here.

- Collaboration Tools
- Email
- Change Tracking Tools
- Version Control Tools
- Document Generation Tools

Collaboration tools are noted to have been 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 project use such software. Many of the projects are located on GitHub, and it's 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. They often use GitHub for this, and a few use Git or CVS. Document generation tools are mentioned in the artifacts of 12 of the projects. Tools such as Sphinx and Doxygen are explicitly used in this capacity. Drasil could be considered as an alternative to generate artifacts since it is geared specifically towards research software domains like LBM software.

7.3 Principles, Processes, and Methodologies - DONE 1st DRAFT

Most of the assessed 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 that shortening 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 listed categories during our assessment, for the categories that were measured.

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 these packages that development involved following any specific model, like a waterfall of agile development model. One developer noted that while no formal model is used, their development model is something similar to a combination of agile and waterfall development models. The developers teams of the LBM packages are fairly small and easily organized without the need for such methodologies.

Seven of the software packages contain artifacts outlining the general development process, like basic instructions on how to contribute. Most of the development teams accept outside contributors, but generally the teams are tight-knit, often working at the same in-

stitution.

The developers that were interviewed all noted similar project management processes. In teams of only a couple of developers, the addition of new features or major changes are discussed with the entire team. Projects with more than a couple of developers have lead developer roles. These lead developers review potential additions to the software. One of the developers noted in their interview that a peer review process is used to review major changes and additions.

The software packages frequently use GitHub for managing the project. Typically there are several development branches and changes from these branches are merged into the master branch.

Documentation was also noted as playing a significant role in the development process, specifically with on-boarding new developers. An intention 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 12 software packages, and was noted during interviews with developers. The mentioned document generation tools are Sphinx and Doxygen.

Two types of software changes were noted during interviews. One is feature additions, which arise from a scientific or functional need. The development of these changes was noted to involve more formal discussions within the development team and the execution involves lead developers. 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 trackers are 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 examining the artifacts. The use of automatic installation processes is also common. Most often this involved a Make script.

7.4 Pain Points - DONE 1st Draft

Developers were asked to comment on obstacles in their development process, obstacles encountered by user, and potential future obstacles with the software packages.

Several development process obstacles were noted. One developer noted that their small development team, common in LBM software packages, has a lack of time to implement new features. 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.

A lack of software development experience was noted by the developer of a third package where many of the team members on the project are domain experts. There is a steep learning curve to get these team members to contribute good quality source code. This has been somewhat addressed as the code is now generally written in a way that the developer thinks makes it as easy as possible for contributions.

The developer also noted that there is a lack of incentive and funding in academia for developing good scientific software that people end up using. There are no journals that publish such source code. There is no place for the development of good quality software. Scientific software is often developed by the researchers that temporarily use it in their own research and keep the development and use in-house. This software often does not meet the standards that would be required by external users. Sometimes the software is commercialized but it is not open source.

Documentation is important, but it's quality could be improved. There is often no time or funding for maintaining quality documentation for software that is rarely used externally.

Furthermore, documentation generally does not teach the underlying metaphysics. It sometimes provides an introduction to these topics. Users would need to already be familiar with these topics or spend time referencing suitable textbooks. The documentation generally focuses on explaining how to use the software. It is not feasible for package documentation to address the underlying metaphysics topics in detail. Sometimes frequently asked questions on the underlying theory are answered in the documentation, or pointers to answers are noted.

Technically, setting up parallelization was also noted as a pain point for one of the developers, and the introduction of continuous integration by another. Software development knowledge could mitigate some pain points. As already noted, many of the developers are domain experts and not professional software developers. An interviewed developer with more extensive software development experience noted that eliminating equivalent statements using macros had helped improve the quality of their source code, specifically helping with extending code to run on both the CPU and GPU.

Difficulties with ensuring correctness were also noted by several developers. They indicated that tests are run on new additions,on both the individual modules and on the entire system to verify continued system correctness. These tests compare the package output to known correct output using test cases. A developer commented that the amount of testing data that is needed for some cases is a problem, as free services do not offer capabilities to store and frequently process such large amounts of data and in-house testing solutions needed to be created.

A few obstacles related to users were found. Several developers noted that users sometimes try to combine LBM methods with cases incorrectly, and that this is not theoretically sound. Some users think that the packages will work out of the box to solve their cases, while the packages require a good understanding of computational fluid dynamics and significant effort in setting up each case. These packages are not like more commercial software pack-

ages, they are generally set up to solve specific research problems. While they are modifiable to solve similar problems, these modifications are not trivial. So far this 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 explain theoretical principles, as well as test user input, 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.

One developer commented that parts of their source code had been refactored to Python partially to address usability. Python was perceived as a much more usable language that would be easy for future users and developers to learn and understand. According to this developer this was one of the biggest steps in terms of addressing usability.

A few potential future obstacles were noted. One developer said that their source code had been written with a specific application in mind and that there was too much coupling between components. This would have an impact on future modifiability and reusability when trying to extend the software. The effected 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 and provided.

7.5 Quality Recommendations - DONE 1st Draft

The following points should be considered to improve LBM software quality.

7.5.1 Installability

- Include OS compatibility, including versions.

- Include complete installation instructions.
- Include all dependencies in the installation instructions, required versions, and how to install them.
- If possible, automate the installation of dependencies.
- Installation instructions should be written as if the user does not have any dependencies installed and is installing on a clean OS.
- Limit installation instruction location to one location.
- Automate the installation process as much as possible.
- Include descriptive error messages for errors encountered during installation.
- Include a way to validate the installation.
- Include instructions for uninstalling the software.

7.5.2 Surface Correctness and Verifiability

- Use a requirements specification document.
- Make public the requirements specification document, or explicitly reference the domain theory that the software is designed from.
- Ensure the above information is easy to find. Consider adding it to the user manual.
- Development teams should include both domain experts and experiences software developers.
- Use and make public detailed documentation. Consider using automatic document generation tools.

- Include detailed tutorials with expected output.
- Use unit testing during development. Make public the facilities for unit testing.
- Modularize the source code.
- Use continuous integration tools and processes during development.

7.5.3 Surface Reliability

- Include descriptive error messages where appropriate.
- In case automatic installation of dependencies fails, the system should indicate to the user what dependencies are needed and the installation procedure should be able to skip installing the dependencies after they have been manually installed.
- The packages should include detailed tutorials, including dependencies, expected output, and supplementary documentation.

7.5.4 Surface Robustness

- Include descriptive error messages for cases of unexpected input including incorrect data types, empty input, and missing files or links.

7.5.5 Surface Performance

- Implement using parallelization tools. Consider GPU processing, CUDA, and MPI.
- The user should be able to process the models on either the CPU or GPU.

7.5.6 Surface Usability

- Include user hardware and software requirements.

- Include a user tutorial with expected output.
- Include a detailed user manual. Identify elements of user interfaces.
- Explicitly state the boundaries of the software, and intended applications.
- Identify appropriate fluid dynamics problems that can be solved by the software.
- Provide background domain information or a reference to it.
- Explicitly state the requirements and commitments required of the user to properly model their problem.
- Document expected user characteristics.
- Keep documentation in one location.
- Maintain a user support model (Git, email, forum, FAQ)
- If possible, consider using user-friendly scripting languages like Python. Especially in parts of the software that are likely to be modified or reviewed by users.

7.5.7 Maintainability

- Include high quality artifacts. Ideally most of the common and less common artifacts listed in Section 7.1.
- Include version numbers and release notes for all major releases.
- Have a defined process for accepting contributions.
- Include documentation for making contributions to the project.
- Use an issue tracker (Git, email, SourceForge, other) to manage bugs and changes. Issues should be closed regularly.

- Use a version control system (GitHib, CVS).
- Source code needs to be well commented.
- Ensure source code is modular.
- Eliminate code duplication.
- If possible, develop using languages that are easy to read and quick to learn.

7.5.8 Modifiability

- Ensure a high degree of code modularity and abstraction.
- If possible, develop using languages that are easy to read and quick to learn.
- Consider future modifiabilility in the design stage prior to any development.
- Consider flexibility of data structures and data storage in the design stage.
- Make public validation benchmarks.

7.5.9 Reusability

- Modularize the source code.
- Document module interfaces.
- Provide API documentation, if applicable.

7.5.10 Surface Understandability

- Maintain consistent formatting style across source code files.
- Identify a coding standard, if one is used during development.

- Use consistent, distinctive, and meaningful code identifiers. Identify algorithms that are used.
- Hard code domain constants.
- Add meaningful comments. They should indicate what is being done in each section of source code.
- Modularize the source code.
- Include user guides and usage requirements.
- In the documentation identify the limits of the software.
- Identify the cases that the software can be applied to.
- Note the requirements of the user in order to set up their case. Identify what parameters are needed.
- Consider adding a FAQ section to the documentation.

7.5.11 Traceability

- Update all relevant documentation when a change to the software is made.
- Use automatic document generation tools to limit the time spent on updating documentation.
- Include documentation aimed at developers, like a developers guide.
- Provide documentation that orients users and developers to artifacts, the source code, and system architecture.
- Highlight new system features between artifact versions.

7.5.12 Visibility and Transparency

- Identify the development model that was used, if one was used.
- Identify the development environment.
- Identify the development process and how to contribute.
- Include notes with all releases.
- Write modular code.
- Communicate all contributions and changes to the entire development team.
- Use continuous integration processes and tools.
- Consider peer review processes for contributions.
- Decrease code duplication.
- Maintain updated developer documentation.
- Make use of project management software.

7.5.13 Reproducibility

- Compare results of methods against manually calculated or known correct results.
- Automate the above comparison. Run the comparison after code changes.
- Consider peer reviews for code contributions.
- Maintain detailed and up to date design documentation.

7.5.14 Unambiguity

- Documentation must include all technical requirements and dependencies for using the software.
- Documentation should detail how to properly use the software.
- Documentation should outline correct and incorrect use cases of the software.
- Documentation should either explain underlying CFD theories or reference appropriate resources.
- Consider asking users for feedback on the documentation.

7.6 Designation Comparison

answers last question of research questions:

How does software designated as high quality by this methodology compare with top rated software by the community?

8 Conclusion

- key findings restated - comment on future work in state of the practice SCS

9 Appendix

9.1 Research Questions

The following are the research questions that are considered.

1. What artifacts are present in current software packages?
2. What tools (development, dependencies, project management) are used by current software packages?
3. What principles, processes, and methodologies are used in the development of current software packages?
4. 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?
5. For research software developers, what specific actions are taken to address the following:
   (a) installability
   (b) correctness and verifiability
   (c) reliability
   (d) robustness
   (e) performance
   (f) usability
   (g) maintainability
   (h) modifiability
   (i) reusability
   (j) understandability
   (k) traceability
   (l) visibility and transparency
   (m) reproducibility
   (n) unambiguity
6. How does software designated as high quality by this methodology compare with top rated software by the community?

9.2 Measurement Template

The table below lists the set of measures that are used to assess each software product. The first set identifies summary information, followed by 9 sets for software qualities and 3 sets for raw metrics. Each measure is followed by the type for a valid result. A superscript indicate that a response of this type needs to be accompanied by explanatory text.

Table 3: Measurement Template

Summary Information
Software name? (string)
URL? (URL)
Affiliation (institution(s)) (string or N/A)
Software purpose (string)
Number of developers (all developers that have contributed at least one commit to the project) (use repo commit logs) (number)
How is the project funded? (unfunded, unclear, funded*) where * requires a string to say the source of funding
Initial release date? (date)
Last commit date? (date)
Status? (alive is defined as presence of commits in the last 18 months) (alive, dead, unclear)
License? (GNU GPL, BSD, MIT, terms of use, trial, none, unclear, other*) * given via a string
Platforms? (set of Windows, Linux, OS X, Android, other*) * given via string
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)
Development model? (open source, freeware, commercial, unclear)
Publications about the software? Refers to publications that have used or mentioned the software. (number or unknown)
Source code URL? (set of url, n/a, unclear)
Programming language(s)? (set of FORTRAN, Matlab, C, C++, Java, R, Ruby, Python, Cython, BASIC, Pascal, IDL, unclear, other*) * given via string
Is there evidence that performance was considered? Performance refers to either speed, storage, or throughput. (yes, no)
Additional comments? (can cover any metrics you feel are missing, or any other thoughts you have)

Installability (Measured via installation on a virtual machine.)
Are there installation instructions? (yes, no)
Are the installation instructions in one place? Place referring to a single document or web page. (yes, no, n/a)
Are the installation instructions linear? Linear meaning progressing in a single series of steps. (yes, no, n/a)
Are the instructions written as if the person doing the installation has none of the dependent packages installed? (yes, no, unclear)
Are compatible operating system versions listed? (yes, no)
Is there something in place to automate the installation (makefile, script, installer, etc)? (yes*, no)
If the software installation broke, was a descriptive error message displayed? (yes, no, n/a)
Is there a specified way to validate the installation? (yes*, no)
How many steps were involved in the installation? (Includes manual steps like unzipping files) Specify OS. (number, OS)
What OS was used for the installation? (Windows, Linux, OS X, Android, other*) *given via string
How many extra software packages need to be installed before or during installation? (number)
Are required package versions listed? (yes, no, n/a)
Are there instructions for the installation of required packages / dependencies? (yes, no, n/a)
Run uninstall, if available. Were any obvious problems caused? (yes , no, unavail)
Overall impression? (1 .. 10)
Additional comments? (can cover any metrics you feel are missing, or any other thoughts you have)

Correctness and Verifiability

Any reference to the requirements specifications of the program or theory manuals? (yes , no, unclear)

What tools or techniques are used to build confidence of correctness? (literate programming, automated testing, symbolic execution, model checking, assertions used in the code, Sphinx, Doxygen, Javadoc, confluence, unclear, other*) * given via string

If there is a getting started tutorial? (yes, no)

Are the tutorial instructions linear? (yes, no, n/a)

Does the getting started tutorial provide an expected output? (yes, no*, n/a)

Does your tutorial output match the expected output? (yes, no, n/a)

Are unit tests available? (yes, no, unclear)

Is there evidence of continuous integration? (for example mentioned in documentation, Jenkins, Travis CI, Bamboo, other) (yes*, no, unclear)

Overall impression? (1 .. 10)

Additional comments? (can cover any metrics you feel are missing, or any other thoughts you have)

Surface Reliability

Did the software break during installation? (yes , no)

If the software installation broke, was the installation instance recoverable? (yes, no, n/a)

Did the software break during the initial tutorial testing? (yes, no, n/a)

If the tutorial testing broke, was a descriptive error message displayed? (yes, no, n/a)

If the tutorial testing broke, was the tutorial testing instance recoverable? (yes, no, n/a)

Overall impression? (1 .. 10)

Additional comments? (can cover any metrics you feel are missing, or any other thoughts you have)

Surface Robustness

Does the software handle unexpected/unanticipated input (like data of the wrong type, empty input, missing files or links) reasonably? (a reasonable response can include an appropriate error message.) (yes, no)

For any plain text input files, if all new lines are replaced with new lines and carriage returns, will the software handle this gracefully? (yes, no, n/a)

Overall impression? (1 .. 10)

Additional comments? (can cover any metrics you feel are missing, or any other thoughts you have)

Surface Usability
Is there a getting started tutorial? (yes, no)
Is there a user manual? (yes, no)
Are expected user characteristics documented? (yes, no)
What is the user support model? FAQ? User forum? E-mail address to direct questions? Etc. (string)
Overall impression? (1 .. 10)
Additional comments? (can cover any metrics you feel are missing, or any other thoughts you have)
Maintainability
What is the current version number? (number)
Is there any information on how code is reviewed, or how to contribute? (yes*, no)
Are artifacts available? (List every type of file that is not a code file for examples please look at the Artifact Name column of https://gitlab.cas.mcmaster.ca/SEforSC/se4sc/-/blob/git-svn/GradStudents/Olu/ResearchProposal/Artifacts_MiningV3.xlsx) (yes*, no, unclear) *list via string
What issue tracking tool is employed? (set of Trac, JIRA, Redmine, e-mail, discussion board, sourceforge, google code, git, BitBucket, none, unclear, other*) * given via string
What is the percentage of identified issues that are closed? (percentage)
What percentage of code is comments? (percentage)
Which version control system is in use? (svn, cvs, git, github, unclear, other*) * given via string
Overall impression? (1 .. 10)
Additional comments? (can cover any metrics you feel are missing, or any other thoughts you have)
Reusability
How many code files are there? (number)
Is API documented? (yes, no, n/a)
Overall impression? (1 .. 10)
Additional comments? (can cover any metrics you feel are missing, or any other thoughts you have)

Surface Understandability (Based on 10 random source files)
Consistent indentation and formatting style? (yes, no, n/a)
Explicit identification of a coding standard? (yes, no, n/a)
Are the code identifiers consistent, distinctive, and meaningful? (yes, no , n/a)
Are constants (other than 0 and 1) hard coded into the program? (yes, no , n/a)
Comments are clear, indicate what is being done, not how? (yes, no , n/a)
Is the name/URL of any algorithms used mentioned? (yes, no , n/a)
Parameters are in the same order for all functions? (yes, no , n/a)
Is code modularized? (yes, no , n/a)
Overall impression? (1 .. 10)
Additional comments? (can cover any metrics you feel are missing, or any other thoughts you have)
Visibility/Transparency
Is the development process defined? If yes, what process is used. (yes, no, n/a)
Are there any documents recording the development process and status? (yes, no))
Is the development environment documented? (yes, no)
Are there release notes? (yes, no)
Overall impression? (1 .. 10)
Additional comments? (can cover any metrics you feel are missing, or any other thoughts you have)
Raw Metrics (Measured via git_stats)
Number of text-based files. (number)
Number of binary files. (number)
Number of total lines in text-based files. (number)
Number of total lines added to text-based files. (number)
Number of total lines deleted from text-based files. (number)
Number of total commits. (number)
Numbers of commits by year in the last 5 years. (Count from as early as possible if the project is younger than 5 years.) (list of numbers)
Numbers of commits by month in the last 12 months. (list of numbers)

Raw Metrics (Measured via scc)
Number of text-based files. (number)
Number of total lines in text-based files. (number)
Number of code lines in text-based files. (number)
Number of comment lines in text-based files. (number)
Number of blank lines in text-based files. (number)
Repo Metrics (Measured via GitHub)
Number of stars. (number)
Number of forks. (number)
Number of people watching this repo. (number)
Number of open pull requests. (number)
Number of closed pull requests. (number)

9.3 Grading Template

The table below lists how each quality measure of the measurement template is used to calculate an overall impression in each software quality set.

Table 4: Grading Template

Installability (Measured via installation on a virtual machine.)
Are there installation instructions? (yes=1, no=-1)
Are the installation instructions in one place? Place referring to a single document or web page. (yes=1, no=0, n/a=0)
Are the installation instructions linear? Linear meaning progressing in a single series of steps. (yes=1, no=0, n/a=0)
Are the instructions written as if the person doing the installation has none of the dependent packages installed? (yes=1, no=0, unclear=0)
Are compatible operating system versions listed? (yes=1, no=0)
Is there something in place to automate the installation (makefile, script, installer, etc)? (yes*=1, no=-1)
If the software installation broke, was a descriptive error message displayed? (yes=0, no=-2, n/a=1)
Is there a specified way to validate the installation? (yes*=1, no=0)
How many steps were involved in the installation? (Includes manual steps like unzipping files) Specify OS. (<10 = 1)
What OS was used for the installation? (does not count)
How many extra software packages need to be installed before or during installation? (<10 = 1)
Are required package versions listed? (yes=1, no=0, n/a=1)
Are there instructions for the installation of required packages / dependencies? (yes=1, no=0, n/a=1)
Run uninstall, if available. Were any obvious problems caused? (yes=0, no=1, unavail=1)
Overall impression? (a sum of >10 is rounded down to 10)

Correctness and Verifiability
Any reference to the requirements specifications of the program or theory manuals? (yes=2, no=0, unclear=0)
What tools or techniques are used to build confidence of correctness? (any=1, unclear=0)
If there is a getting started tutorial? (yes=2, no=0)
Are the tutorial instructions linear? (yes=1, no=0, n/a=0)
Does the getting started tutorial provide an expected output? (yes=1, no*=0, n/a=0)
Does your tutorial output match the expected output? (yes=1, no=0, n/a=0)
Are unit tests available? (yes=1, no=0, unclear=0)
Is there evidence of continuous integration? (for example mentioned in documentation, Jenkins, Travis CI, Bamboo, other) (yes*=1, no=0, unclear=0)
Surface Reliability
Did the software break during installation? (yes=0, no=5)
If the software installation broke, was the installation instance recoverable? (yes=5, no=0, n/a=0)
Did the software break during the initial tutorial testing? (yes=0, no=5, n/a=0)
If the tutorial testing broke, was a descriptive error message displayed? (yes=2, no=0, n/a=0)
If the tutorial testing broke, was the tutorial testing instance recoverable? (yes=3, no=0, n/a=0)
Surface Robustness
Does the software handle unexpected/unanticipated input (like data of the wrong type, empty input, missing files or links) reasonably? (a reasonable response can include an appropriate error message.) (yes=5, no=0)
For any plain text input files, if all new lines are replaced with new lines and carriage returns, will the software handle this gracefully? (yes=5, no=0, n/a=5)
Surface Usability
Is there a getting started tutorial? (yes=3, no=0)
Is there a user manual? (yes=4, no=0)
Are expected user characteristics documented? (yes=1, no=0)
What is the user support model? FAQ? User forum? E-mail address to direct questions? Etc. (one=1, two+=2, none=0)

Maintainability
What is the current version number? (provided=1, nothing=0)
Is there any information on how code is reviewed, or how to contribute? (yes*=1, no=0)
Are artifacts available? (List every type of file that is not a code file for examples please look at the Artifact Name column of https://gitlab.cas.mcmaster.ca/SEforSC/se4sc/-/blob/git-svn/GradStudents/Olu/ResearchProposal/Artifacts_MiningV3.xlsx) (Rate 0 2 depending on how many and perceived quality)
What issue tracking tool is employed? (nothing=0, email of other private=1, anything public or accessible by all devs (eg git) = 2)
What is the percentage of identified issues that are closed? (50%+=1, <50%=0)
What percentage of code is comments? (10%+=1, <10%=0)
Which version control system is in use? (anything=2, nothing=0)
Reusability
How many code files are there? (0-9=0, 10-49=1, 50-99=3, 100-299=4, 300-599=5, 600-999=6, 1000+=8)
Is API documented? (yes=2, no=0, n/a=0)
Surface Understandability (Based on 10 random source files)
Consistent indentation and formatting style? (yes=1, no=0, n/a=0)
Explicit identification of a coding standard? (yes=1, no=0, n/a=0)
Are the code identifiers consistent, distinctive, and meaningful? (yes=2, no=0, n/a=0)
Are constants (other than 0 and 1) hard coded into the program? (yes=1, no=0, n/a=0)
Comments are clear, indicate what is being done, not how? (yes=2, no=0, n/a=0)
Is the name/URL of any algorithms used mentioned? (yes=1, no=0, n/a=0)
Parameters are in the same order for all functions? (yes=1, no=0, n/a=0)
Is code modularized? (yes=1, no=0, n/a=0)
Visibility/Transparency
Is the development process defined? If yes, what process is used. (yes=3, no=0, n/a=0)
Are there any documents recording the development process and status? (yes=3, no=0))
Is the development environment documented? (yes=2, no=0)
Are there release notes? (yes=2, no=0)

9.4 Ethics Approval

This project received ethics clearance from the McMaster Research Ethics Board on February 20, 2021.

Project Title: AIMSS - State of the Practice

MREB#: 5219

9.5 Developer Interview Questions

Information about these interview questions: This gives you an idea what I would like to learn about the development of domain software. Interviews will be one-to-one and will be open-ended (not just yes or no answers). Because of this, the exact wording may change a little. Sometimes I will use other short questions to make sure I understand what you told me or if I need more information when we are talking such as: So, you are saying that ?), to get more information (Please tell me more?), or to learn what you think or feel about something (Why do you think that is?).

1. Interviewees current position/title? degrees?
2. Interviewees contribution to/relationship with the software?
3. Length of time the interviewee has been involved with this software?
4. How large is the development group?
5. Do you have a defined process for accepting new contributions into your team?
6. What is the typical background of a developer?
7. What is your estimated number of users? How did you come up with that estimate?
8. What is the typical background of a user?
9. Currently, what are the most significant obstacles in your development process?
10. How might you change your development process to remove or reduce these obstacles?
11. How does documentation fit into your development process? Would improved documentation help with the obstacles you typically face?
12. In the past, is there any major obstacle to your development process that has been solved? How did you solve it?
13. What is your software development model? For example, waterfall, agile, etc.
14. What is your project management process? Do you think improving this process can tackle the current problem? Were any project management tools used?
15. Was it hard to ensure the correctness of the software? If there were any obstacles, what methods have been considered or practiced to improve the situation? If practiced, did it work?
16. When designing the software, did you consider the ease of future changes? For example, will it be hard to change the structure of the system, modules or code blocks? What measures have been taken to ensure the ease of future changes and maintains?

17. Provide instances where users have misunderstood the software. What, if any, actions were taken to address understandability issues?

18. What, if any, actions were taken to address usability issues?

19. Do you think the current documentation can clearly convey all necessary knowledge to the users? If yes, how did you successfully achieve it? If no, what improvements are needed?

20. Do you have any concern that your computational results wont be reproducible in the future? Have you taken any steps to ensure reproducibility?

References

Iso/iec/ieee international standard - systems and software engineering–vocabulary. *ISO/IEC/IEEE 24765:2017(E)*, pages 1–541, 2017. doi: 10.1109/IEEESTD.2017.8016712.

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

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

F. Benureau and N. Rougier. Re-run, Repeat, Reproduce, Reuse, Replicate: Transforming Code into Scientific Contributions. *ArXiv e-prints*, August 2017.

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

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

Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of software engineering*. Prentice Hall PTR, 1991.

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

Benjamin Graille and Loïc Gouarin. pylbm documentation. 2017.

Alan Gray and Kevin Stratford. 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.

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

IEEE. Ieee standard glossary of software engineering terminology. Standard, IEEE, 1991.

IEEE. Recommended practice for software requirements specifications. *IEEE Std 830-1998*, pages 1–40, October 1998. doi: 10.1109/IEEESTD.1998.88286.

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

ISO/IEC. Systems and software engineering - systems and software quality requirements and evaluation (square) - system and software quality models. Standard, International Organization for Standardization, Mar 2011.

Panagiotis Kalagiakos. The non-technical factors of reusability. In *Proceedings of the 29th Conference on EUROMICRO*, page 124. IEEE Computer Society, 2003.

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

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

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

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

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

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

David Lorge Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, (1):1–9, 1976.

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

Spencer Smith. Systematic development of requirements documentation for general purpose scientific computing software. In *14th IEEE International Requirements Engineering Conference (RE'06)*, pages 209–218. IEEE, 2006.

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

W. Spencer Smith, Adam Lazzarato, and Jacques Carette. State of practice for mesh generation software. *Advances in Engineering Software*, 100:53–71, October 2016.

W. Spencer Smith, Adam Lazzarato, and Jacques Carette. State of the practice for GIS software. https://arxiv.org/abs/1802.03422, February 2018a.

W. Spencer Smith, Yue Sun, and Jacques Carette. Statistical software for psychology: Comparing development practices between CRAN and other communities. https://arxiv.org/abs/1802.07362, 2018b. 33 pp.

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

Ian Sommerville. *Software Engineering 9*. Pearson Education, 2011.

Richard H Thayer and Merlin Dorfman. Ieee recommended practice for software requirements specifications. *IEEE Computer Society, Washington, DC, USA, 2nd ed. edition*, 2000.

Axel Van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Proceedings fifth ieee international symposium on requirements engineering*, pages 249–262. IEEE, 2001.

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

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

Yuzhi Zhao. Automated knowledge extraction based on a scientific computing software documentation generation framework, 2018.