

STATE OF THE PRACTICE FOR LATTICE BOLTZMANN METHOD SOFTWARE

By
PETER MICHALSKI.

A Report
Submitted to the School of Graduate Studies
in Partial Fulfillment of the Requirements
for the degree
Master of Engineering in Computing and Software

McMaster University
© Copyright by Peter Michalski, July 2021

Abstract

We analyze the state of the practice of software development of Lattice Boltzmann Methods software domain by quantitatively and qualitatively measuring, and comparing, 23 software packages ~~along quality attributes~~. A methodology for assessing the state of the practice of software development in scientific computing software domains is presented. A domain analysis of the Lattice Boltzmann Methods software family is made, and a candidate software package ~~list~~ ^{is pre} is identified and curated. The packages are assessed to answer software development related research questions to understand how software quality is impacted by software development choices, including principles, processes, and tools. Software developers are interviewed to identify development pain points, and to identify how software quality is ensured. Quantitative data is used to rank the software packages using ~~an~~ ^{the} analytical hierarchy process. The ranking designations are compared with rankings from the software development community. Recommendations for improving software along quality metrics are made. This knowledge can be used to guide future development of scientific computing software, specifically along quality attributes, and to reduce software quality failures.

Keywords: Lattice Boltzmann Methods, Scientific Computing, Software Engineering, Software Family, Software Quality

→ Add results — Give a list of packages that were identified to have the highest quality and why (maybe 3 packages?)
— Give a list of the top recommendations

Abstracts should include specific results

Contents

1	Introduction	1
1.1	Research Questions	2
1.2	Motivation	4
1.3	Scope	5
1.4	Organization	6
2	Domain Analysis	7
2.1	Lattice Boltzmann Systems	8
2.2	Commonalities	8
2.2.1	Lattice Boltzmann Method Solvers	10
2.2.2	Input	12
2.2.3	Output	12
2.3	Variabilities	12
2.3.1	Lattice Boltzmann Method Solvers	13
2.3.2	Input	17
2.3.3	Output	17
2.3.4	System Constraints	18
2.4	Parameters of Variation	19
2.4.1	Lattice Boltzmann Method Solvers	20
2.4.2	Input	26
2.4.3	Output	26
2.4.4	System Constraints	27
3	Methodology	29
3.1	Process	30
3.2	Software Qualities	31
3.2.1	Installability	31
3.2.2	Correctness	31
3.2.3	Verifiability	32
3.2.4	Reliability	32
3.2.5	Robustness	33
3.2.6	Performance	33
3.2.7	Usability	33
3.2.8	Maintainability	34
3.2.9	Modifiability	34
3.2.10	Reusability	34
3.2.11	Understandability	35
3.2.12	Traceability	35

3.2.13	Visibility and Transparency	35
3.2.14	Reproducibility	36
3.2.15	Unambiguity	36
3.3	Identify Candidate Software	36
3.4	Filter the Software List	37
3.5	Empirical Measures	40
3.6	Analytical Hierarchy Process	41
4	Results	42
4.1	Quantitative Findings and AHP Results	42
4.1.1	Installability	42
4.1.2	Surface Correctness and Verifiability	46
4.1.3	Surface Reliability	48
4.1.4	Surface Robustness	49
4.1.5	Surface Performance	51
4.1.6	Surface Usability	51
4.1.7	Maintainability	53
4.1.8	Reusability	55
4.1.9	Surface Understandability	58
4.1.10	Visibility and Transparency	59
4.1.11	Overall Quality	61
4.2	Qualitative Findings From Developer Interviews	62
4.2.1	Surface Correctness and Verifiability	62
4.2.2	Surface Usability	63
4.2.3	Maintainability	64
4.2.4	Modifiability	64
4.2.5	Surface Understandability	65
4.2.6	Traceability	66
4.2.7	Visibility and Transparency	67
4.2.8	Reproducibility	69
4.2.9	Unambiguity	70
5	Answers To Research Questions	72
5.1	Artifacts Present	72
5.1.1	Common Artifacts	72
5.1.2	Less Common Artifacts	73
5.1.3	Rare Artifacts	74
5.2	Tools Used	75
5.2.1	Development Tools	76
5.2.2	Dependencies	76

5.2.3	Project Management Tools	77
5.3	Principles, Processes, and Methodologies - DONE 1st DRAFT	78
5.4	Pain Points	80
5.5	Quality Recommendations	85
5.5.1	Installability	85
5.5.2	Surface Correctness and Verifiability	86
5.5.3	Surface Reliability	87
5.5.4	Surface Robustness	87
5.5.5	Surface Performance	87
5.5.6	Surface Usability	88
5.5.7	Maintainability	89
5.5.8	Modifiability	90
5.5.9	Reusability	91
5.5.10	Surface Understandability	91
5.5.11	Traceability	92
5.5.12	Visibility and Transparency	92
5.5.13	Reproducibility	93
5.5.14	Unambiguity	94
5.6	Designation Comparison	94
5.6.1	Repository Ranking Metrics	95
5.6.2	Domain Expert Recommended Software	96
5.7	Threats To Validity	98
6	Conclusion	100
6.1	Highlighted Recommendations	101
6.2	Future State Of The Practice Assessments	102
7	Appendix	104
7.1	Developer Interview Questions	104
7.2	Measurement Template	106
7.3	Grading Template	112
7.4	Eliminated Software Packages	115
7.5	Ethics Approval	116

List of Figures

1	Streamlines of the flow past a stationary circular cylinder at $Re = 10, 20,$ and 40	1
2	Commonly used LBM models in two and three dimensions	23
3	AHP Installability Score	45
4	AHP Surface Correctness and Verifiability Score	47
5	AHP Surface Reliability Score	49
6	AHP Surface Robustness Score	50
7	AHP Surface Usability Score	52
8	AHP Maintainability Score	55
9	AHP Reusability Score	56
10	AHP Understandability Score	59
11	AHP Visibility and Transparency Score	60
12	AHP Overall Score	61

List of Tables

1	Alive Software Packages	38
2	Dead Software Packages	39
3	Git Repository Data	54
4	Module Data	57
5	Repository Ranking Metrics	95
6	Ranking Comparison	97
7	Measurement Template	106
8	Grading Template	112
9	Eliminated Software Packages	115

Reference Material

Software Engineering Related Definitions and Acronyms

AHP: Analytical Hierarchy Process

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

CPU: Central Processing Unit

Goal: Goals capture, at different levels of abstraction, the various objectives the system under consideration should achieve [45].

GPU: Graphics Processing Unit

LOC: Lines of Code

OS: Operating System

OTS: Off-The-Shelf

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 [43].

SCS: Scientific Computing Software

Software Family: A set of programs with an extensive amount of common properties [28].

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

Lattice Boltzmann Related Definitions and Acronyms

1D 1-Dimensional

2D: 2-Dimensional

3D: 3-Dimensional

BGK: Bhatnagar-Gross-Krook [5]

CFD: Computational Fluid Dynamics

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.

1 Introduction

Computational Fluid Dynamic (CFD)

We analyze the development of software packages that use Lattice Boltzmann Methods (LBM), a class of computational fluid dynamics (CFD) methods, for fluid simulation. These LBM algorithms consider the behaviours of a collection of particles as a single unit of an intermediate size, and predict their positional probability as they move through a lattice structure. Figure 1 from [8] presents the streamlines of converged flow past a stationary circular cylinder with varied Reynolds number, a common task of LBM.

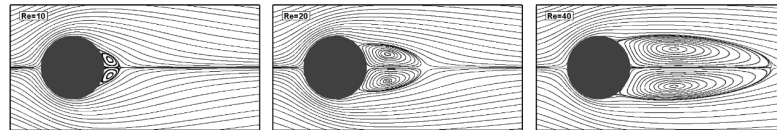


Figure 1: Streamlines of the flow past a stationary circular cylinder at $Re = 10, 20$, and 40 [8]

LBM have several advantages over conventional CFD methods, including a simple calculation procedure, improved parallelization, and robust handling of complex geometries [9]. LBM are further detailed in Section 2.

Scientific Computing Software (SCS) is defined as the use of computer tools to analyze or simulate mathematical models of real world systems [32]. Given the importance of such software, scientists and engineers desire methods and tools to sustainably develop it with high quality. This has led to the formation of groups like the Software Sustainability Institute (SEI) and Better Scientific Software (BSS).

The goal of this report is to analyze the current state of development of LBM SCS to provide insight into its best practices and to offer guidance for future development. We want to understand how software quality in the LBM SCS family is impacted by software

development choices, including principles, processes, and tools.

In this report red text denotes internally linked sections of this document. Cyan text denotes an external URL link. Green text denotes a link to the bibliography.

1.1 Research Questions

The LBM software packages are assessed to answer the following research questions:

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?


The packages are assessed for a given set of quality attributes using quantitative and qualitative data. The qualities are defined in Section 3.2. Qualitative data is gathered from interviews with software package developers. The interview questions are listed in the Appendix in 7.1. Quantitative data is measured using the quality metrics listed in the Appendix in 7.2. In this context the purposes of the report are as follows:

- Develop and test an updated methodology for assessing the state of the practice of SCS projects
- Describe the specific example of LBM software structure and goals
- Report on the measure of a subset of LBM software packages along quality metrics

Rather than point to the Appendix, point to the section where you discuss the interview (That section can point to the Appendix)


Rather than point to the Appendix, point to the section where you discuss measuring quality - that section can point to the Appendix

You listed the qualities above

- 
- Evaluate the state of the practice of LBM software development along quality attributes
 - Make suggestions for improving LBM software along quality attributes
 - Make suggestions for improving state of the practice of SCS assessments

1.2 Motivation

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 to 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 [39], Mesh Generators [38], Oceanographic Software [36], Seismology software [41], and statistical software for psychology [40].

In the course of this assessment we updated the methodology that was used during previous assessments. Details of the new methodology are in [33]. The previous set of research questions were critically assessed and modified. The updated questions are listed in Section 1.1. In this re-boot we collected more quantitative and qualitative data, focused on software quality attributes, and added 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.

1.3 Scope

We analyze a filtered set of LBM software packages along quantitative and qualitative 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; they may not represent the true quality of the software. Surface measurements are taken as they allow us to apply the same measurement, with reasonable effort, along all software packages despite technical and functional variabilities among the set of software packages. Not all qualities are quantitatively measured. For example, performance is not measured by running the code. A surface investigation of the documentation of each software package for performance information is conducted instead. The measured software packages are open source, and it is recommended to keep this in mind when considering the recommendations of this report. Some recommendations may not apply to closed-source software. 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 3.2. 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 differ among software packages due to their inherent organizational and technical differences.

1.4 Organization

The report is organized as follows:

- **Introduction** to the report, including its purpose, motivation, scope, and organization.
- LBM software **Domain Analysis**. A domain analysis consists of systematically identifying and documenting the commonalities, variabilities, and terminology of a software family [47]. The purpose of this analysis in this state of the practice assessment is to better classify the different software products based on their functionality.
- **Methodology** of this state of the practice assessment, including the steps of the methodology, software quality definitions and how these qualities are assessed in this report, an overview of candidate software selection, empirical measures gathered, and the analytical hierarchy process. This follows a general methodology for assessing the state of the practice for SCS domains.
- Quantitative and qualitative **Results**.
- An analysis of the results and **Answers To Research Questions**.
- **Conclusion** to the report including comments on suggestions for future state of the practice assessments.
- **Appendix**: This section includes our Research Questions, the Measurement Template, Grading Template, Ethics Approval, and Developer Interview Questions.

2 Domain Analysis

✓ A domain analysis consists of systematically identifying and documenting the commonalities, variabilities, and parameters of variation of a software family [47]. A software family is defined by [28] 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.

➤ We added this domain commonality and variability analysis to our original methodology because the first time we did the State of the Practice exercise we found we weren't comparing "apples to apples". Software packages can have considerable variation, even when the packages appear to be in the same software family. Viewing the ranking exercise as a means for selecting the right tool for a job, functionality is important. A 2D Lattice Boltzmann solver is not a substitute for a 3D Lattice Boltzmann solver when the latter is required. Therefore, this exercise seeks to classify the different software products based on their functionality. ✓ Nonfunctional qualities are assessed through other means in our process. For instance, through usability experiments as noted in the [Methodology for Assessing the State of the Practice for Domain X](#) document.

This domain analysis of the LBM SCS family is organized into the four subsection located below. The first subsection reviews the basics of Lattice Boltzmann systems. The next three subsections consist of lists of commonalities, variabilities, and parameters of variation, respectively. These three sections form the heart of the domain analysis and include an extensive set of cross-references to demonstrate the relationships between the different items.

2.1 Lattice Boltzmann Systems

LBM are a family of fluid dynamics algorithms for simulating single-phase and multi-phase fluid flows, often incorporating additional physical complexities [7]. They consider the behaviours of a collection of particles as a single unit at the mesoscopic scale, between the nanoscopic and microscopic scales. 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.

LBS model fluid dynamics within a boundary using a predefined lattice structure and 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 [2]. There are many standardized lattice models - individual solvers within the family might only use a subset of them. LBM use 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.

2.2 Commonalities

This section lists common features among 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, generate the simulation, output the results. Section 2.2.1 describes the commonalities for the simulation step. Section

2.2.2 highlights the input information that is required for Lattice Boltzmann systems. The next section, Section 2.2.3, 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 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 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 individual software packages, their simulation models, inputs, and outputs.

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.

I think
you
removed
this
part



2.2.1 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. Collision operators map collisions of particles within the lattice space. The Bhatnagar-Gross-Krook Collision Operator is a common LBM collision operator that preserves continuity for a discretized model, for each velocity direction i . Its equation is $\Omega_i = \frac{1}{\tau}(f_i^{eq} - f_i)$, where τ is the relaxation rate towards equilibrium, f^{eq} is the equilibrium particle probability distribution function, and f is the particle probability distribution function.
Related Variability	V5 V6

Item Number	C3
Description	All Lattice Boltzmann versions use 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. The equation is $(\frac{d}{dt} + \mathbf{e} \cdot \nabla_{\mathbf{x}} + \frac{\mathbf{F}}{m} \cdot \nabla_{\mathbf{e}})f = \Omega(t)$, where \mathbf{e} is the velocity, $\nabla_{\mathbf{x}}$ is the position vector gradient, $\nabla_{\mathbf{e}}$ is the velocity gradient, and $\Omega(t)$ is the collision operator as a function of time.
Related Variability	V8

what are \mathbf{F} , m and f ?

2.2.2 Input

Item Number	C6
Description	The LBS require fluid, model, and boundary information for the problem. This includes, but is not limited to, fluid acceleration rate, velocity, and viscosity, as well as the number of dimensions in the lattice model, and the number of velocity directions in the lattice.
Related Variability	V12

2.2.3 Output

Item Number	C7
Description	LBS write fluid predictions, such as the output of the transport equation, to memory.
Related Variability	V13 V14

2.3 Variabilities

This section provides a list of characteristics that may vary among family members. As in Section 2.2, the first three subsections on variabilities are organized into the following sublists: Simulation Models, 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 three headings provided for each variability are: Description, Related Commonality, and Related Parameter. 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.

2.3.1 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 used with the distribution function can vary. These are based on the number of velocity directions in the model.
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. Is this the number of fluids
Related Commonality	C6 a multi fluid simulation.
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 various conditions.
Related Commonality	C1
Related Parameter	P11

2.3.2 Input

Item Number	V12
Description	The input interface can vary between LBS.
Related Commonality	C6
Related Parameter	P12

2.3.3 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

2.3.4 System Constraints

Item Number	V15
Description	Hardware that processes the calculations
Related Commonality	None
Related Parameter	P15

Item Number	V16
Description	Operating systems on which LBS run.
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

2.4 Parameters of Variation

This section specifies the parameters of variation for the variabilities listed in Section 2.3. They are organized into the same five subcategories as employed previously: Simulation Models, 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 2.3. 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~~^{build} 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 could be associated with the presence of a domain specific language that would allow postponing decisions on the values of the parameter of variation.

2.4.1 Lattice Boltzmann Method Solvers

Item Number	P1
Corresponding Variability	V1
Range of Parameters	OpenMP, OpenCL, CUDA, MPI are used if the execution of the LBS is parallelized.
Binding Time	Build Time

Item Number	P2
Corresponding Variability	V2
Range of Parameters	Equilibrium approximation varies between incompressible or compressible models.
Binding Time	Build Time

Item Number	P3
Corresponding Variability	V ³
Range of Parameters	Various data structures can be used to store function output, including single and multi-dimensional arrays, depending on the problem model and developer preferences.
Binding Time	Build Time

Item Number	P4
Corresponding Variability	V ⁴
Range of Parameters	Numerous coefficients for equilibrium distribution function based on number of velocity directions. The number of velocity directions is typically 2, 3, 5, 9, 13, 15, 19, or 27.
Binding Time	Build Time

Item Number	P5
Corresponding Variability	V5
Range of Parameters	LBS model has 1, 2, or 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. Figure 2 from [44] illustrates two commonly used LBM models in two and three dimensions.
Binding Time	Build Time or Run Time

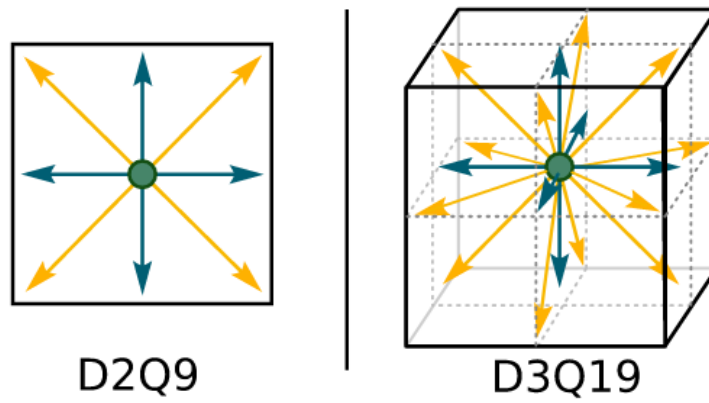


Figure 2: Commonly used LBM models in two and three dimensions

Item Number	P7
Corresponding Variability	V ⁷
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. LBS sometimes use one or the other, often the collision transport equation. A Boolean parameter could be used to select between these equations in systems that can apply either equation.
Binding Time	Build Time

Item Number	P9
Corresponding Variability	V9
Range of Parameters	LBS can model a natural number of 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. All of these parameters have the type of real number.
Binding Time	Build Time

Item Number	P11
Corresponding Variability	V11
Range of Parameters	Lattice boundary can have reflective or non-reflective conditions. Some LBS will only model reflective or non-reflective conditions. If there is a choice then this can be indicated by a Boolean parameter.
Binding Time	Build Time

2.4.2 Input

Item Number	P12
Corresponding Variability	V12
Range of Parameters	Input can be graphical, text or file.
Binding Time	Build Time

2.4.3 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

2.4.4 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.
Binding Time	Run Time

3 Methodology

In this project we set out to answer the research questions listed in Section 1.1 for the LBM SCS domain. The process involved systematically measuring and analyzing members of this software family along the quality attributes found in Section 3.2. This included gathering quantitative and qualitative measurements. A goal of the project was to produce a quality assessment for LBM software packages.

The methodology used in this LBM software assessment is a general state of the practice methodology that can be applied to any SCS domain. It was developed as an update to previous state of the practice exercises. ~~This general methodology is linked in the next subsection.~~

We collected quantitative data using the measures found in the measurement template of Section 7. Some of this was empirical software engineering related data, such as the number of files, number of lines of code (LOC), percentage of issues that are closed, etc. Most of the data was gathered by manually investigating the software, its source code, and its artifacts, while some was gathered using the empirical measurement tools discussed in Section 3.5.

We also collected qualitative data by interviewing the software package developers and asking them the questions found in the Appendix in 7.1. Furthermore, we solicited the assistance of domain experts to better assess each software package by leveraging their experience to assess the functional and non-functional requirements for the software domain.

This section begins by describing the steps of the overall process we used to select, measure and compare LBM software. This is followed by quality definitions and how these qualities were assessed 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 that were used, and the ~~analytic hierarchy~~ ^{AHP} process (AHP).

3.1 Process

The following steps provide an overview of how the assessment was conducted:

1. List candidate software packages for the domain. This is discussed in Section 3.3.
2. Filter the software package list. This is discussed in Section 3.4.
3. Gather the source code and documentation for each software package.
4. Collect empirical measures. This is discussed in Section 3.5.
5. Measure using the measurement template. This is discussed in Section 3.5. The measurement template can be found in the Appendix in ^A7.2.
6. Survey the developers. The developers of each software package in the filtered software list were contacted for voluntary interviews. The interview questions can be found in the Appendix in ^A7.1.
7. Use AHP to rank the software packages. This is discussed in Section 3.6.
8. Analyze the results and answer the research questions. The answers can be found in Section 5.

These steps are further detailed in the [Methodology for Assessing the State of the Practice for Domain X](#) document.

3.2 Software Qualities

Software quality attributes facilitate the measurement and comparison of software packages in this state of the practice assessment. We adopt software quality definitions from various researchers and subject matter expert entities. Some of the definitions are from [34]. The quality measurement results are found in Section 4 of this report. Section 5.5 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.

3.2.1 Installability

Installability is measured by the effort required for the installation, uninstallation or reinstallation of a software product in a specified environment [19] [23]. 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 also has a positive effect on the measure of installability. Similarly, the ease of uninstallation has an affect on the measure of installability.

3.2.2 Correctness

A software program is correct if it behaves according to its stated specifications [12]. This requires that the specification is available. Software is unlikely 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.

3.2.3 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 [42]. Similarly to correctness, verifiability is correlated with the availability of a specification and with reference to domain knowledge. A good measure of verifiability is further correlated with the availability of well written tutorials that include expected output, with software unit testing documentation, and with evidence of continuous integration during the development process.

3.2.4 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) [12] [26]. Reliability is thus positively correlated with the absence of errors during installation and use. Recoverability from errors also improves reliability.

3.2.5 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 [6] [11]. A good measure of robustness correlates with a reasonable reaction to unexpected input, including data of the wrong type, empty input, or missing files or links. A reasonable reaction includes an appropriate error message and the ability to recover the system.

3.2.6 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) [16] [48]. In this state of the practice assessment performance was not quantitatively measured. Instead the documentation of each software package was observed for information that alludes to a consideration of performance, such as parallelization tools.

3.2.7 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 [27]. A good measure of usability correlates with the presence of documentation, including tutorials, manuals, and defined user characteristics, and user support. Preferably the user support model has avenues to contact developers and report issues.

3.2.8 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 [16] [6]. 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 artifacts can greatly decrease the effort needed to modify software. There are many documentation artifacts that can improve maintainability, including user and developer manuals, specifications, README files, change logs, release notes, publications, forums, and instructional websites.

3.2.9 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 [1]. 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.

3.2.10 Reusability

Reusability refers to the extent to which components of a software package can be used with or without adaptation in other software packages [20]. A good measure of reusability results from a large number of easily reusable components. Increased software modularization, defined as the presence of smaller components with well defined interfaces, is important. For this state of the practice assessment, a good measure of reusability correlates

with an increased number of code files, and the availability of API documentation.

3.2.11 Understandability

Understandability is measured by the capability of the software package to enable the user to understand its suitability and function [18]. It is an artifact-dependent quality. Understandability is different for the user-interface, source code, and documentation. In this state of the practice analysis, understandability focuses on the source code. It is measured by the consistency of a formatting style, the extend of modularization, the explicit identification of coding standards, the presence of meaningful identifiers, and clarity of comments.

3.2.12 Traceability

Traceability refers to the ability to link the software implementation and the software artifacts, especially the requirement specification [25]. 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.

3.2.13 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 [11]. 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.

3.2.14 Reproducibility

Software achieves reproducibility if another developer can take the requirements documentation and re-obtain the same software artifacts [4]. This includes the output of the software, where the scientific results are compared between software implementations, or between software implementations and manually calculated results. 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 won't be reproducible in the future, and if they have taken any steps to ensure reproducibility.

3.2.15 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 [17]. 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 to achieve it.

3.3 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.

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

didate software:
 why we missed? We don't have to add it now, but it would be good to know whether one was missing

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

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

3.4 Filter the Software List

To reduce the number of members in the candidate software list to a manageable size, the following filters were applied. The filters were applied in the priority order listed.

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

For the third item in the above filter, software packages were characterized as ‘alive’ if their related documentation had been updated within the last 18 month. Packages were

categorized as ‘dead’ if the last update of this information was more than 18 month ago.

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

Table 1: Alive Software Packages

While the initial list had 45 packages, filtering by scope, usage, and age decreased the size of the list to 23 packages. Many of the 22 packages that were removed could not be tested as there was no installation guide, they were incomplete, source code was not publicly available, a license was needed, or the project was out of scope or not up to a standard that would support incorporating them into this study. These eliminated software packages are listed in the Appendix in Section 7.4. Of the remaining 23 packages that were studied, some were kept on the list despite being marked as dead due to their prevalence on authoritative lists on LBM software and due to their surface excellence, specifically the considerable time that was put into these projects.

Can you be
specific about which were kept?

The final list of software packages that were analyzed in this project can be found in the following two tables. Table 1 lists packages that fell into the ‘alive’ category as of mid 2020, and Table 2 lists packages that were ‘dead’ at that time.

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

Table 2: Dead Software Packages

There is considerable variation among these software packages, including their intended purpose, size, user interfaces, and software languages used. For example, the OpenLB software package is predominantly a C++ package that makes use of hybrid parallelization and was designed to address a range of CFD problems [15]. The software package pyLBM is an all-in-one Python language package for numerical simulations [13]. ESPResSo is an extensible simulation package that is specifically for research on soft matter, and is written in C++ and Python [46]. The HemeLB package is used for efficient simulation of fluid flow in several medical domains, and is written predominantly in C, C++, and Python [24].

3.5 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 1.1.

All of the quality measurements that are part of the AHP analysis are empirical measurements. Qualitative data gathered during interviews with developers is not part of the AHP analysis. 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 the freeware tools discussed below. The data that was gathered is listed in the measurement template found in the Appendix in [7.2](#). 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 the Appendix in [7.3](#), the output of which was analyzed using the AHP described in Section 3.6.

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.

Are these github
measurements
summarized
somewhere in
the results section?

3.6 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 the Appendix in [A2](#) using the grading template found in the Appendix in [A3](#). AHP performs a pairwise analysis using a matrix and generates an overall score as well as individual quality scores for each software package. [\[38\]](#) 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 includes requirements and usage information.

You don't want to start a sentence like this.

Say

Smith Et al (2016) [\[38\]](#) shows... [\[38\]](#).

4 Results

This section presents the quantitative and qualitative findings from data that was gathered for each quality listed in Section 3.2. Quantitative results are presented in Section 4.1, and qualitative results are presented in Section 4.2.

4.1 Quantitative Findings and AHP Results

This subsection presents the quantitative findings from data that was gathered using the measurement template found in Section 7.2. The results of the AHP analysis based on that data are also presented in this subsection.

4.1.1 Installability

All of the 23 software packages that were tested have installation instructions. As noted previously, many of the 23 software packages that were part of the original long list of 45 packages were removed due to not including documentation or installation instructions.

All 23 packages on the short list can be installed on some Unix-like systems. Seven packages could be installed on Windows, and five on macOS. Operating system compatibility is found in the documentation of 19 software packages. All but one of the software packages, TCLB, were tested on Ubuntu for this state of the practice assessment. TCLB was tested on CentOS, since this operating system is mentioned in its installation instructions.

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, like with Ludwig, incomplete installation instructions are found on a home page, with more detailed

instructions located on another web-page, or within the documentation. Maintainability and correctness of these instructions could be improved if all the instructions were in one location.

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

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

About half of the installation instructions are written as if the person doing the installation has none of the dependent packages installed. It is common for software packages to not list all of their dependencies despite listing some. This was the case for many packages, including ESPResSo++, Ludwig, and LUMA. Sometimes only an error message during the installation process informs the user of the requirement of these additional packages. A detailed rewrite of the installation instructions from the point of view of installation on a clean operating system is suggested. A clean environment can be achieved for testing purposes by using a virtual machine. ✓

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

Sixteen software packages have less than 10 manual installation steps. If dependencies are installed in one command then none of the software packages take more than 20 steps to install. The average number of steps is about eight, and the fewest is two (LB3D-Prime).

All but six (ESPResSo, HemeLB, laboetie, LB3D-Prime, lbmpy, waLBerla) of the software packages have a way to specifically verify the installation. Most have some sort of tutorial examples that can be run by the user. Some other ways of installation validation include validation scripts (LB2D-Prime, lettuce, Ludwig, LUMA), automatic validation after the installation (LatBo.jl), and instructions to manually review the file system (LIMBES).

Uninstallation instructions were found for only one of the software packages, pyLBM. ✓

Figure 3 shows the installability ranking of the software packages using AHP. Software packages with a higher score (MechSys, Palabos, ESPResSo, OpenLB) tend to have one set of linear installation instructions that are written as if the person doing the installation has none of the dependencies installed. The instructions often list compatible operating system versions and include instructions for the installation of dependencies. The top ranked packages often incorporate some sort of automation of the installation process and have fewer manual installation steps. The number of dependencies a package has does not correlate with a higher score. The ability to validate the installation process, often through

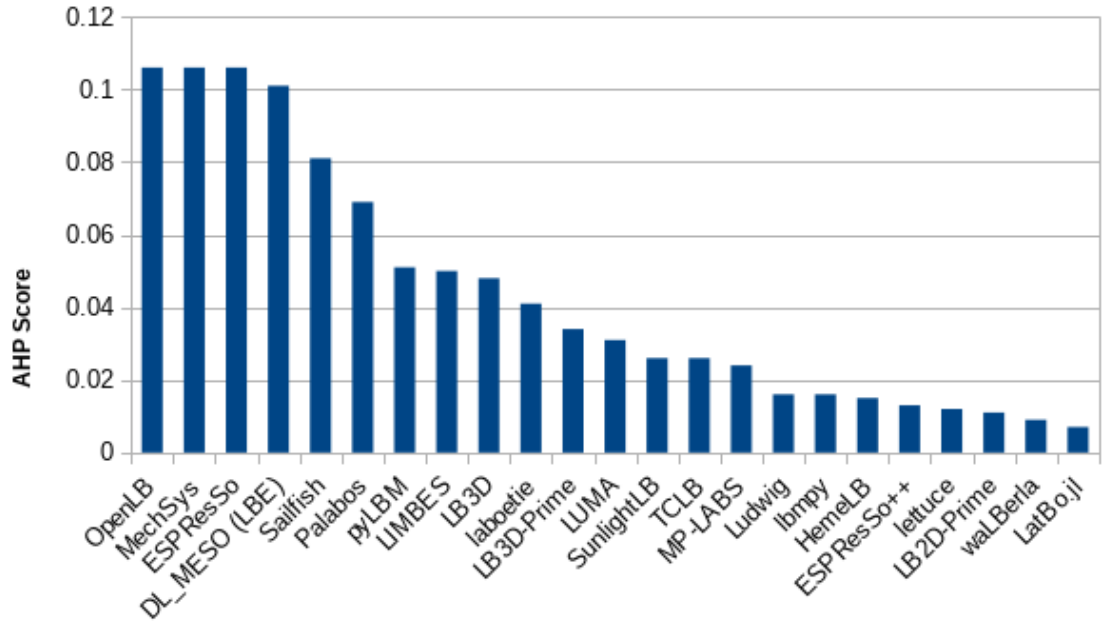


Figure 3: AHP Installability Score

tutorials or test examples that include expected output, is correlated with a higher score. Furthermore, the top six ranked packages are noted as being alive.

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

4.1.2 Surface Correctness and Verifiability

Sixteen of the software packages include a requirements specification artifact or explicitly reference domain theory, often only the latter. Software packages that distribute requirements specification information, such as DL_MESO (LBE), generally keep it brief and include it within other documentation. This artifact is often found within a user manual, on a web-page, or is mentioned in related publications. In the latter case the user may need to spend significant time to find this information.

Document generation tools are explicitly used by 12 software packages. Sphinx is used by eight of them, and Doxygen is used by seven. Several of the packages use both.

Tutorials are available for 18 of the software packages. Generally they are linearly written and easy to follow. However, only eight tutorials provide an expected output. It is not possible to verify the correctness of the output of the software packages that are missing this key information. In these cases the user may need to assume correctness if there are no visible errors.

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

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

Figure 4 shows the surface correctness and verifiability ranking of the software packages using AHP. Software packages with a higher score tend to have a visible requirements

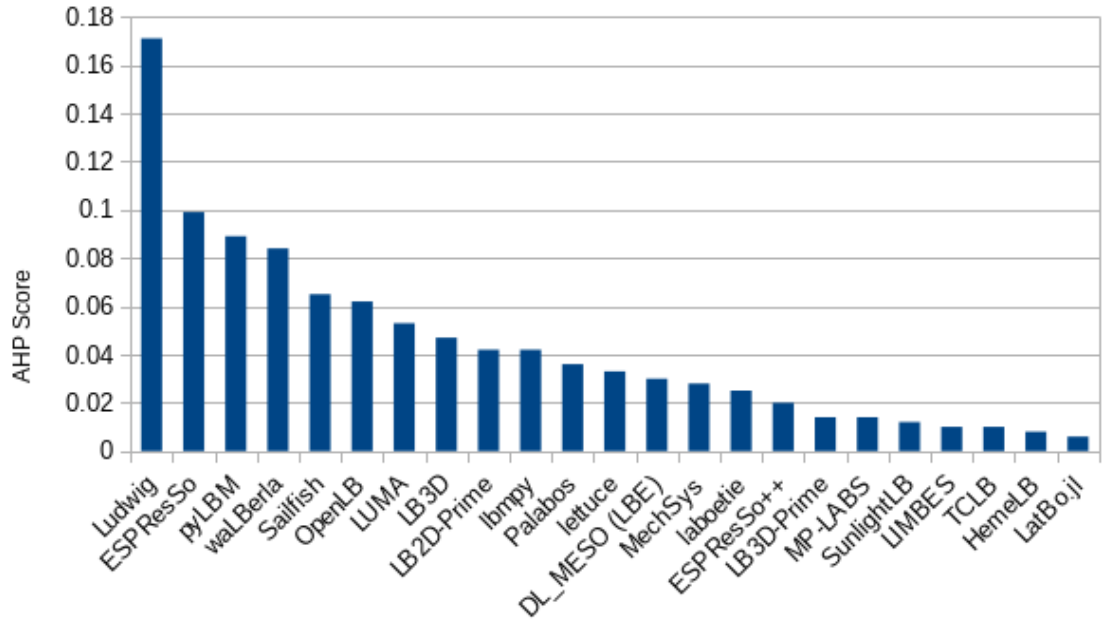


Figure 4: AHP Surface Correctness and Verifiability Score

specification or references to theory documentation. They also explicitly use at least one document generation tool that builds confidence of correctness. The top ranked software packages all include an easy to follow getting started tutorial, and most of these include expected output. Only the top ranked package, Ludwig, provided unit testing. It and the second ranked package, ESPResSo, explicitly incorporated continuous integration in the development process. Furthermore, eight of the top 10 ranked packages are noted as being alive.

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

techniques such as Bamboo, Jenkins, and Travis CI, help verify correctness.

4.1.3 Surface Reliability

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

Of the 13 software packages that installed correctly and also have tutorials, four (pyLBM, ESPResSo++, LIMBES, Ludwig) broke during tutorial testing. All of these instances resulted in an error message being displayed. One error (pyLBM) was due to a missing tutorial dependency, another (Ludwig) was due to an invalid command despite following the tutorial, and the final two errors were vague execution errors. Of the four broken tutorial instances, only the one that was missing a dependency was recoverable.

Figure 5 shows the surface reliability ranking of the software packages using AHP. Software packages with a high score either did not break during installation, or the broken installation was recoverable. All of the top five ranked packages have tutorials. One of these packages, pyLBM, broke during tutorial testing, but a descriptive error message helped in

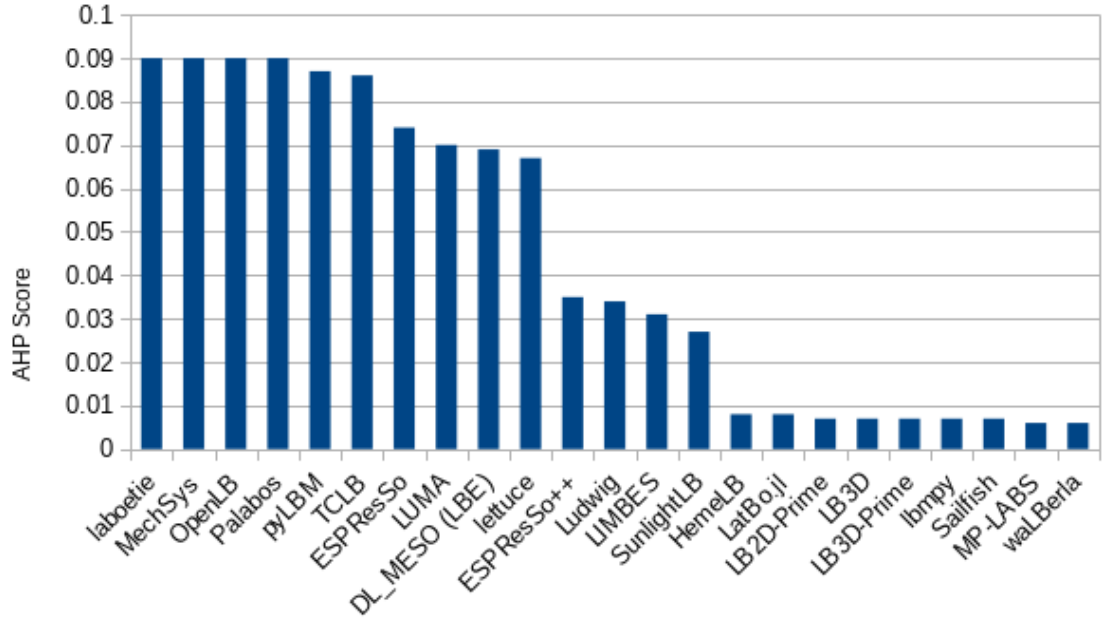


Figure 5: AHP Surface Reliability Score

recovery. Furthermore, nine of the top 10 ranked packages are noted as being alive.

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

4.1.4 Surface Robustness

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

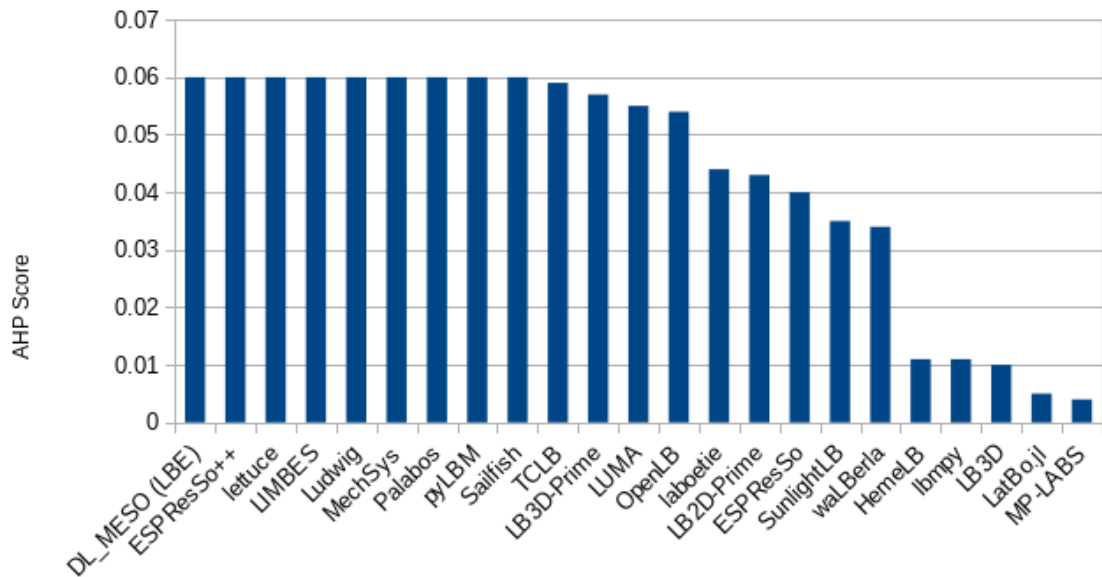


Figure 6: AHP Surface Robustness Score

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

4.1.5 Surface Performance

Although the software packages all apply LBM to solve scientific computing problems, the packages focus on varied CFD problems, with varying parameters, and are technically different from each other. Due to this, a comparison of performance is not appropriate. In this project we instead looked through each software package's artifacts for evidence that performance was considered. The artifacts of 17 software packages mentioned parallelization. This included GPU processing and the CUDA parallel computing platform, which were mentioned in the artifacts of 6 packages (ESPResSo, lbmpy, lettuce, pyLBM, Sailfish, TCLB). GPUs provide superior processing power and speed compared to CPUs, and are often used for scientific computing when a large amount of data is involved. The software package TCLB is implemented in a highly efficient multi-GPU code to achieve performance suitable for model optimization [29]. 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 [14]. While one software package (Sailfish) required CUDA and GPU processing, some (ESPResSo, lbmpy, lettuce, pyLBM, TCLB) have the option of using either the GPU or the CPU. The packages that require GPU and CUDA have better performance at the expense of installability and surface reliability.

4.1.6 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 eight include an expected output. Most user manuals are in the form of a file that can be downloaded, while some are rendered on a web-page. Some packages (waLBerla) do not

How many have both?

have a user manual, but do have useful documentation distributed on their web-pages. Expected user characteristics are documented in four software packages (laboetie, LIMBES, Ludwig, Palabos). Users are typically scientists or engineers. Their background is often physics, chemistry, biophysics, or mathematics. All but one of the packages (LIMBES) have a user support model, and many of them have multiple avenues of user support. The most popular avenue of support is Git, followed by email and forums. One software package (OpenLB) has an FAQ page.

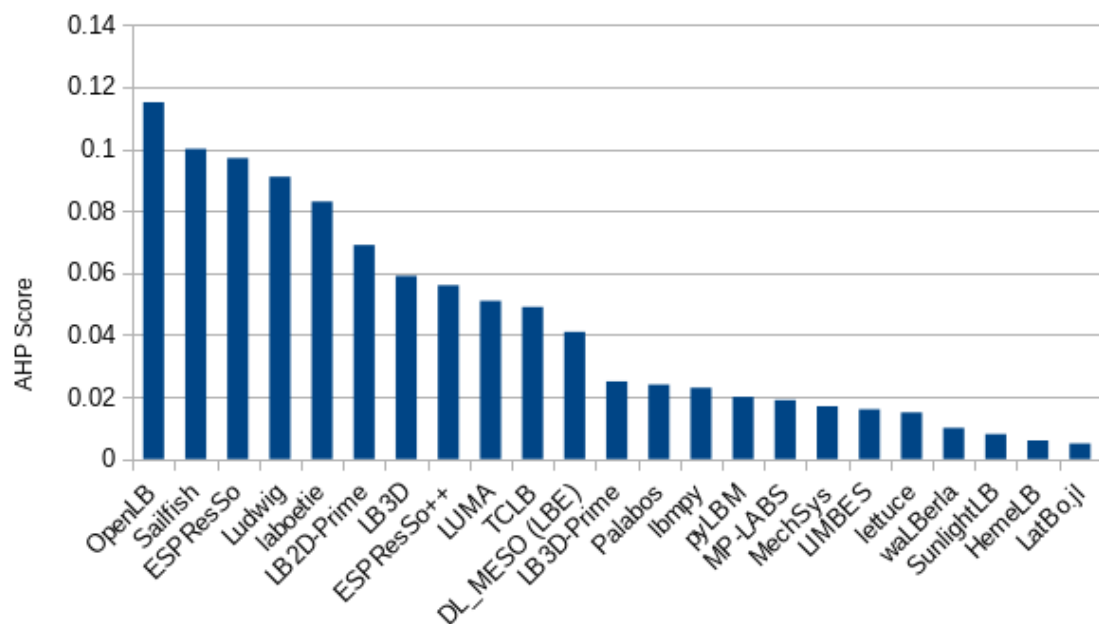


Figure 7: AHP Surface Usability Score

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

as being alive.

4.1.7 Maintainability

Software packages were reviewed for the presence of artifacts. Every type of artifact or file that is not a code file was recorded. The software packages were also reviewed for software release and documentation version numbers. This information could be used to better troubleshoot issues and organize documentation. All but three software packages (LatBo.jl, LB3D-Prime, MechSys) 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, 11 software packages have this information, which was found in various artifacts, including in developer guides, contributor guides, user guides, developer web-pages, and README files.

Issue tracking is used in 22 software packages, 15 of which use Git, six use email, and one (SunlightLB) uses SourceForge. Most software packages that use Git have most of their issues closed, and only three (laboetie, lettuce, Sailfish) have less than 50 percent of their issues closed. Alive packages (11 use Git issue tracking) have 64% of their issues closed, while dead packages (3 use Git issue tracking) have 71% of their issues closed. This information is presented in Table 3. Furthermore, 13 packages that use Git for issue tracking use GitHub as a version control system, while two (Palabos, waLBerla) use GitLab. Of the other packages, one package (SunlightLB) uses CVS for issue tracking, and seven packages do not appear to use any issue tracking system.

Software package code files were further measured for the percentage of code that is comments. The findings are presented in Table 3. 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 code comments is about 14.%, ✓

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

Are there any conclusions to be drawn from this table?
Do these results correlate with any other qualities?

very low

Table 3: Git Repository Data

Figure 8 shows the maintainability ranking of the software packages using AHP. Software packages with a high score provide version numbers on documents and source code

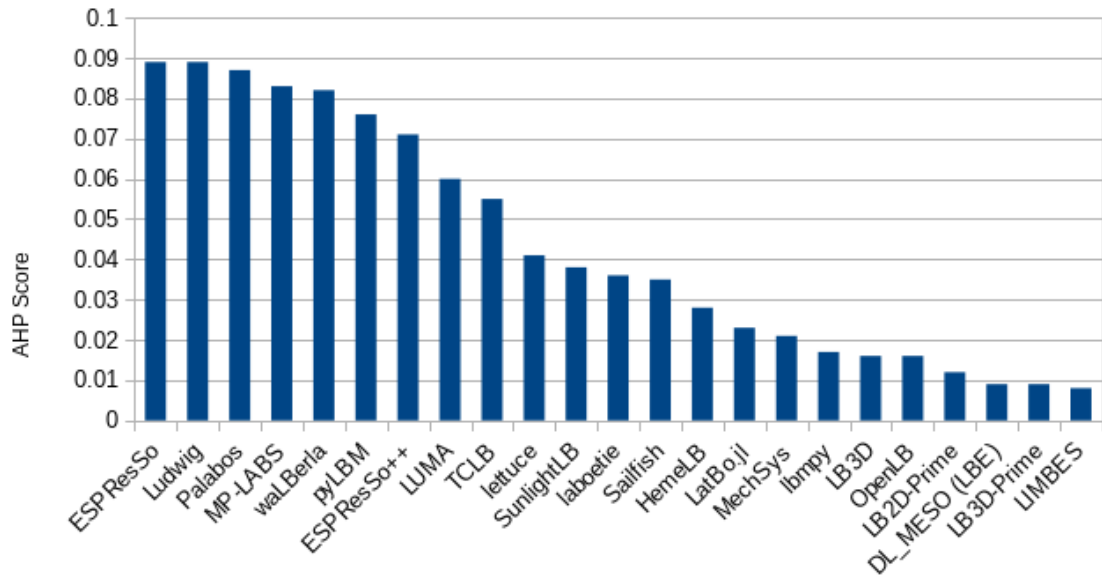


Figure 8: AHP Maintainability Score

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. Furthermore, nine of the top 10 ranked packages are noted as being alive. — which one isn't?

4.1.8 Reusability

Each software package was measured for the total number of source code files. A larger number of source files was associated with increased reusability due to increased modularization. Some packages have more features than others, consequently contributing to reusability since they have more source code that can be reused. The software packages were also reviewed for the presence of API documentation, which indicates that a software

package was developed with interaction between other software applications in mind.

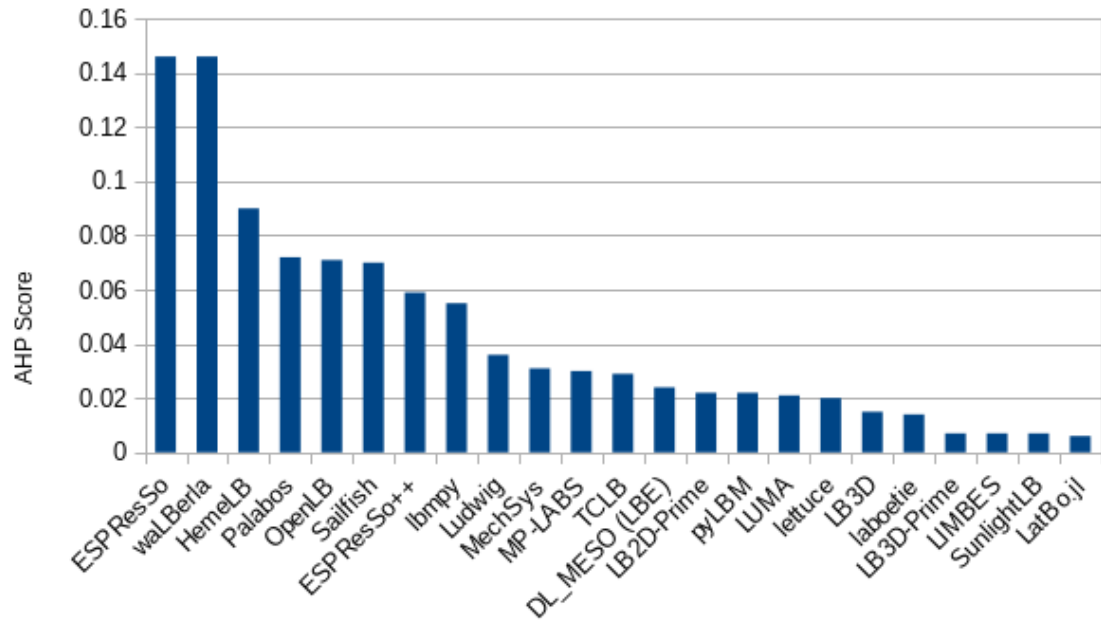


Figure 9: AHP Reusability Score

Figure 9 shows the reusability ranking of the software packages using AHP. Software packages with a high score have thousands of source code files and API documentation. The highest scoring packages, ESPResSo and waLBerla, have extensive functionality, including graphical visualizations as well as modeling that does not use LBM. For this reason a comparison with other software packages is not on a level field. However, these packages do have an abundance of reusable components. Furthermore, nine of the top 10 ranked packages are noted as being alive. — which one is it?

Table 4 shows file and line of code data of the software packages. Packages with a high reusability score do not have many LOC per text file, generally having a few hundred lines or less. This suggests that the source code of these packages is likely functionally

modularized, and modules could be reused in other projects.

Name	Text Files	Binary Files	LOC	Avg. LOC / Text File
DL_MESO (LBE)	310	51	170223	549
ESPResSo	1309	86	186700	143
ESPResSo++	5328	66	969196	182
HemeLB	1065	48	95104	89
laboetie	133	1	48403	364
LatBo.jl	41	0	42172	1029
LB2D-Prime	82	19	54755	668
LB3D	99	76	39766	402
LB3D-Prime	23	6	12944	563
lbmpy	201	28	46489	231
lettuce	62	0	5529	89
LIMBES	26	1	4872	187
Ludwig	859	32	109811	128
LUMA	312	19	4370670	14000
MechSys	324	3	85543	264
MP-LABS	307	3	43124	140
OpenLB	1104	5	209034	189
Palabos	1829	67	547623	299
pyLBM	258	85	32314	125
Sailfish	632	11	69398	110
SunlightLB	36	1	7646	212
TCLB	535	7	43226	81
waLBerla	2395	67	848146	353

Table 4: Module Data

There was a strong focus on modularity when designing the waLBerla framework to enhance productivity, reusability, and maintainability [3]. Its software design has enabled

N.2
table

✓

Is there any other github data that we collect and don't summarize?
(stars? number of contributors etc?)

(might help with the last research question)

waLberla to be successfully applied in several projects as a basis for various extensions [3].

4.1.9 Surface Understandability

For Ten random source code files of each software package were reviewed for several measures. This assessment of surface understandability may not perfectly reflect each package due to the limitation of only ^{practical} testing 10 files ^{being able to test}.

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 four packages (LB2D-Prime, LB3D-Prime, LIMBES, MP-LABS) appear to use vague identifiers, such as single letters for variables. Hard coded symbolic 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 and the comments clearly indicate what is being done. Domain algorithms are noted in the source code of 11 packages. Table 4 suggests that the software packages are modularized to various degrees. When observing the source code files, it was found that 13 of the packages have a consistent style and order of function parameters.

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

hard-coded constants (i.e. HemeLB) are not a good thing → we want symbolic constants

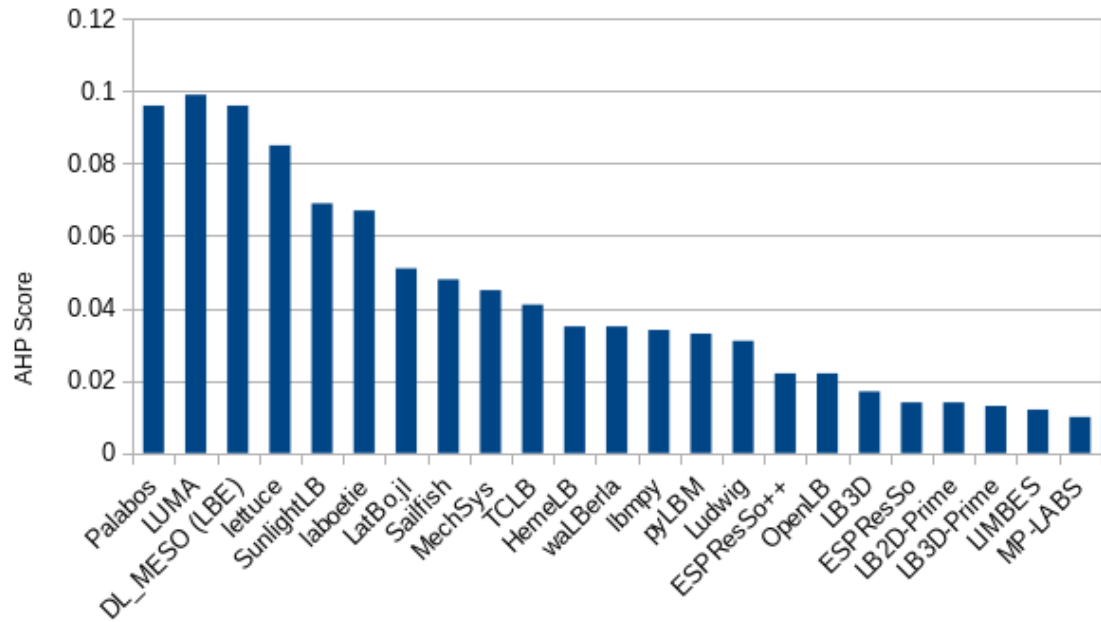


Figure 10: AHP Understandability Score

being alive.

4.1.10 Visibility and Transparency

Software package artifacts were reviewed for the identification of a specific development model, like a waterfall of agile development model, and the presence of documentation recording the development process and standard. They were also reviewed for the identification of the development environment, and the presence of release notes. The packages tended to not explicitly use well-known development models. This was also noted in the interviews with developers, as detailed below. The development teams of these packages are fairly small and easily organized without the need for such processes. Seven of the software packages did have some artifacts outlining the general development process, how to contribute, and the status of the package or its components. Eight of the packages

have artifacts that note the development environment. While this information could help developers, and would improve transparency, the small close-knit nature of the development teams make explicitly publicly specifying this information practically unnecessary. Version release notes were found in nine of the software packages.

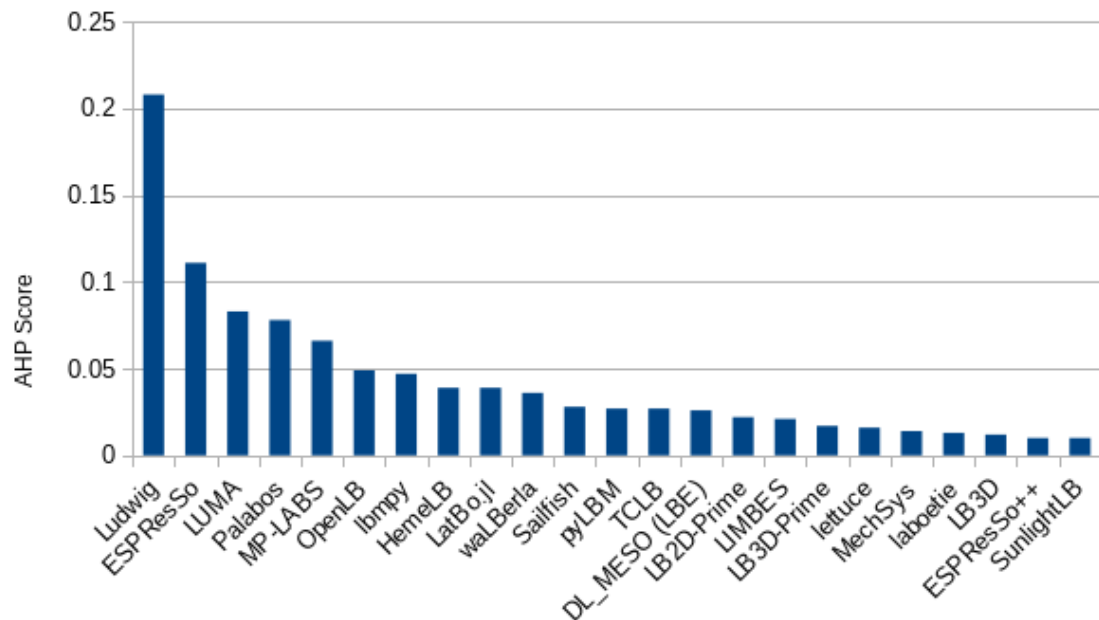


Figure 11: AHP Visibility and Transparency Score

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

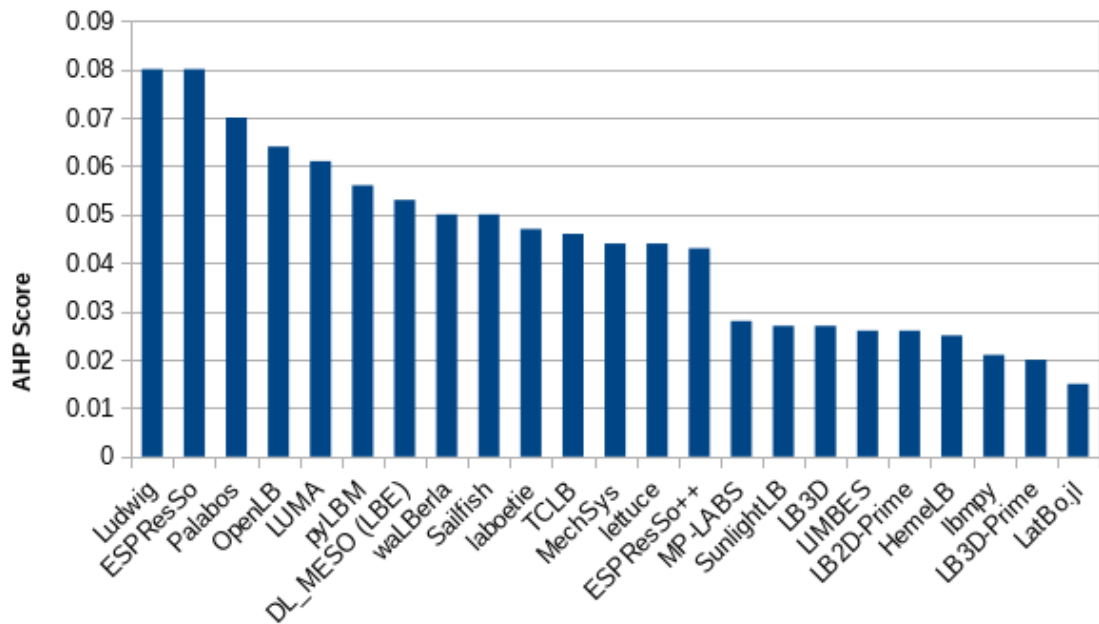


Figure 12: AHP Overall Score

4.1.11 Overall Quality

Figure 12 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.

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

Section 5.5 further analyzes these findings and offers some software quality recommen-

Point out that the overall quality is found with an assumption of an equal weighting of all qualities. If the qualities were ranked differently, the overall quality would change.

dations for future development of LBM software.

4.2 Qualitative Findings From Developer Interviews

This subsection presents the qualitative findings from data that was gathered during interviews with developers. The interview questions are found in Section 7.1.

4.2.1 Surface Correctness and Verifiability

Interviews with developers confirmed that these software packages are developed by domain experts with backgrounds in physics, mathematics and mechanical engineering. It was noted in these interviews that some of the developers do not have formal software engineering education. Some of the development teams include computer scientists. Despite a lack of visible domain documentation and a resulting lower surface correctness and verifiability score, it is clear that some of the software packages were developed by teams with significant domain knowledge on account of the academic backgrounds of their developers.

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

Several interviewed developers alluded to difficulty with testing the correctness of large numbers of features, and some even manually tested program output. The use of well defined unit testing tools could decrease the time spent testing some feature.

4.2.2 Surface Usability

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

Some additional but infrequent software usability issues were commented on by the developers. 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 requirements specifications.

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

4.2.3 Maintainability

Interviews with developers revealed that most projects do have a defined process for accepting contributions from team members. The packages rarely get contributions from outside developers, but the process would be similar as for the aforementioned group.

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 happy to see contribution from outside of their organization, but currently this does not happen.

Furthermore, maintainability has been addressed by increasing source code modularity, reducing duplicate information, and improving abstraction by developing well defined interfaces. This was noted by the developers of ESPResSo and pyLBM. Several software packages have had sections of their code base redeveloped with languages that the developers felt are more understandable and readable, and that are better supported, such as Python. Data structures have also been redeveloped and storage has been improved. A developer of pyLBM mentioned that the geometries and models of their system had been “decoupled”, using abstraction and modularization of the source code, to make it “very easy to add [new] features”.

4.2.4 Modifiability

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

we asked if ease of future changes to the system, modules, and code blocks was considered when designing the software. We also asked if any measures had been taken to ensure the ease of future changes. All of the developers that were interviewed noted that the ease of future changes was considered and that measures to ensure it had been taken, including requiring the separation of software components in the source code architecture.

A high degree of code modularity and abstraction, ensured by separating components and hiding information behind well defined interfaces, was noted by developers as a measure to ensure the ease of future changes. The developer of ESPResSo also noted that some of the code base was transitioned from C to C++, which could ease modifiability of that software package. The developer of TCLB noted that their software package was designed to allow for the addition of some LBM features, but changes to major aspects of the system would be difficult. For example, “implementing a new model will be an easy contribution”, but changes to the “Cartesian mesh...will be a nightmare”. Furthermore, the package was designed with flexible data structures and storage in mind.

Some software packages, like Palabos, provide validation benchmarks for their core fundamental algorithmic ingredients [22]. The stated intent of these benchmarks is to showcase the validity and usefulness of the package to stimulate the development of third-user extensions. The Palabos package identifies as a development framework for modeling problems in various CFD areas.

4.2.5 Surface Understandability

Software developers noted that they believe users have generally found their packages to be understandable. The interviewed developer of ESPResSo commented that some users have attempted to run physically incompatible LBM methods, and the solution was to edit

What words
did they
use to describe
this? This
sounds like how
an SE would
say it

the code to prevent such combinations, as well as to update the documentation to prevent misunderstanding the methods. Similarly, a developer of pyLBM noted that some users had issues setting up parameters for LBM schemes. The solution to this was to update the interface where these parameters are set, as well as to add functionality to test the stability of the parameters. A developer of OpenLB noted that some users lack the background knowledge to easily model fluid dynamics problems using their software. A frequently asked questions section was added to their package website to help users find answers to common questions. The package also has detailed documentation, including guides and usage requirements specification, to better help user understand the software.

4.2.6 Traceability

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

The interviewed developer of ESPResSo noted that all major additions to their package had accompanying changes to artifacts and documentation. They noted that considerable effort had been put into the documentation. They further commented that they want to lower the entry barrier for new developers, and because of that their package has a considerable amount of developer documentation. This documentation informs developers on how to get started, and orients them to the artifacts, source code, and system architecture, as well as how the software package build system works, and how the coupling between the simulation engine and the interface works.

Developers noted the importance of documentation for both the users and developers of their software. New features are always added to the documentation. The developers use

documentation to stay up to date on the status of the software package, and to help expand features, like computational models or algorithms. This is necessary so that the coding standard for these models is kept consistent with new developers.

The importance of documentation for both users and developers was stressed throughout the interviews. However, it was noted several times that a lack of time and funding has a negative affect on the documentation. Most of the developers are scientific researchers evaluated on the scientific papers that they produce. Writing and updating documentation is something that is done in ^{their} free time, if that time arises. Sometimes it is a last priority for the developers. Finding ways to hasten updating documentation would increase the frequency of such updates and benefit both users and developers.

The developer of OpenLB noted the use of documentation generators like Doxygen. It would be advisable for more projects to use such automatic document generation tools, since some projects do not do this.

4.2.7 Visibility and Transparency

Developers were asked to comment on the obstacles in their development process. The developer of ESPReso 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 issue and prevent such issues in the future. Improving the visibility of software changes and a 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

Do you repeat this in the recommendations section?

Our ethics agreement allows us to name the packages right? I'm sure we said that is the correct letter - I just want to verify.

by better updating documentation and ensuring that their contributions are transparent to the rest of the developers. This would help catch issues in the contributions, and improve source code maintenance.

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

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

The developers noted similar project management processes. For 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. The software packages use GitHub for managing the project. Typically there are several development branches as well as the master branch.

4.2.8 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.

The developer of ESPResSo noted a comparison of the results of their methods against manually calculated results. These comparison tests are automatically run for all source code changes. The tests are run when a pull request is opened on GitHub. Even once these tests are complete, a peer review process is 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 frequently compared 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 package is not compared to other output. Depending on the package and how it outputs 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 software specifications and other developer documentation. Some packages are well detailed, and translating the specifications into source code will produce similar results across developers. Developers were not asked to comment on the reproducibility of their source code from their requirements specifications and design documentation. This question should be considered in the next iteration of state of the practice assessments.

4.2.9 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.

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

The developer of pyLBM also noted that their documentation was in reasonable shape,

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

5 Answers To Research Questions



This section answers the research questions listed in Section 1.1 using the quantitative and qualitative data presented in Section 4, additional data from software package repositories and artifacts, and domain expert feedback. Each subsection answers one of the research questions. Software package artifacts are examined in Section 5.1, tools are listed in Section 5.2, development principles, processes, and methodologies are discussed in Section 5.3, development pain points are explored in Section 5.4, and our recommendations for improving software qualities are presented in Section 5.5. A comparison between our designations of the software packages to community rankings is made in Section 5.6. Finally, threats to the validity of this assessment are noted in Section 5.7.

5.1 Artifacts Present

This subsection answers the research question: What artifacts are present in current software packages?

The software packages were examined for the presence of artifacts, which were then categorized by frequency. Below we have grouped them into common, less common, and rare artifacts. Common artifacts were found in 16 to 23 ($>70\%$) of the software packages. Less common artifacts were found in 8 to 15 ($35-70\%$) of the software packages. Rare artifacts were found in 1 to 7 ($<35\%$) of the software packages.

5.1.1 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, OpenLB, and Pala-

bos, have each of these artifacts, except only three of them have a requirements specification or theory notes. Palabos is the only one of these four packages that does not have an artifact from this category.

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

These common artifacts contribute to the quality of the software in the following ways. A list of authors and developers helps potential users and contributors contact project members to answer questions affecting many software qualities. A bug tracker helps with organizing improvements to the software. 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. Licenses promote usage of the software. Requirements specifications, linked theory notes, and related articles and publications help make the software more understandable, decrease ambiguity, and help with verifying its correctness. README files also help with understandability. The tutorials help users become familiar with using the software.

5.1.2 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. At the time of data collection, only one (Palabos) of the four packages did not have a user manual or

guide, but there was a broken link on the package website indicating that such an artifact might exist. (This broken link was later fixed, but this is not reflected in our data. Despite the broken link, Palabos does have a detailed and informative website. Another one (Ludwig) of the top four packages does not appear to have publicly visible design documentation. A third package (OpenLB) from the list does not appear to use a version control system. It is possible that such a system is used by the package since its website notes package version numbers, but the artifacts do not explicitly state the use of such a system.

because it was
not present at the
time of
data
collection)

- 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 with traceability. Functional specifications and notes clarify the software, contributing to usability and understandability. A user manual also helps with those qualities and with installability, and correctness and verifiability. Performance notes suggest that performance was considered when developing the software. Test plans, scripts, and cases, help verify correctness.

5.1.3 Rare Artifacts

The following artifacts were rarely found in the 23 software packages that were assessed. 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 (ESPRESO, Ludwig, Palabos) have information on contributing to the project. Two of them (OpenLB, Palabos) have a FAQ section or forum. One (OpenLB) has verification and validation notes, and a video guide of the software.

- API Documentation
- Developer/Contributor 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 improve 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.

5.2 Tools Used

This subsection answers the research question: What tools (development, dependencies, project management) are used by current software packages?

Software tools are used to support the development, verification, maintenance, and evolution of software, software processes, and artifacts [11]. Many tools are used by LBM software packages. The tools noted here are subdivided into development tools, dependencies, and project management tools.

5.2.1 Development Tools

Development tools support the development of end products, but do not become part of them, unlike dependencies that remain in the application once it is released [11]. 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 in our sources.

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

The above tools can verify the correctness of software during its development. Only two (ESPReso, Ludwig) 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 (Ludwig) explicitly noted the use of proprietary unit testing code written in C. Likewise, the use of proprietary code for verifying the correctness of output was noted by one (pyLBM) of the developers. It is likely that similar tools were used when developing other software packages.

5.2.2 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
- Domain Specific Libraries
- Technical 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 (e.g. Matplotlib, ParaView, Pygame, VTK), data analysis (e.g. Anaconda, Torch), and message passing libraries (e.g. MPICH, Open MPI, PyZMQ). Domain specific libraries are scientific computing libraries (e.g. SciPy). Libraries that are not explicitly stated in artifacts, or were not noted during our observations, may fall outside of these categories.

5.2.3 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
- Version Control Tools
- Email
- Change Tracking Tools
- Document Generation Tools

Collaboration tools are noted as being used when developing the software projects.

Most often email and video conferencing is used. Project management software was not explicitly mentioned, but it is possible that some of the projects use such software. Many of the projects are located on GitHub, and its developers use the platform to help manage their projects, especially bug related issues. Most of the projects appear to use change tracking and version control tools. They often use GitHub for this. One package (lbmpy) uses Git, and another (SunlightLB) uses CVS. Document generation tools are mentioned in the artifacts of 12 of the projects. The tools Sphinx and Doxygen are explicitly used in this capacity.

Only one package uses git?

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

This subsection answers the research question: What principles, processes, and methodologies are used in the development of current software packages?

The points and conclusions in this subsection come from developer interviews and reviews of software package artifacts.

Most of the software packages do not explicitly state in their artifacts the motivations or design principles that were considered when developing the software. One package, Sailfish, indicates in its artifacts that shortening the development time was considered in early stages of design, with the developers opting for using Python and CUDA/OpenCL to achieve this without sacrificing any computational performance. The goals of that project are explicitly listed as performance, scalability, agility and extendability, maintenance, and ease of use. The project scored well in these categories during our assessment.

Processes, like methods, are ways of doing things, especially in an orderly way; while methodologies are defined as systems of methods [11]. It is not explicitly indicated in the artifacts of most of the packages that development involved following any specific model,

like a waterfall or agile development model. One developer (ESPResSo) noted that while no formal model is used, their development model is something similar to a combination of agile and waterfall development models. The developer teams of the LBM packages are fairly small, so it is feasible for them to be organized without the need for such models.

Seven of the software packages contain artifacts outlining the general development process, like basic instructions on how to contribute. Eleven of the packages explicitly convey that they would accept outside contributors, but generally the teams are centralized, often working at the same institution.

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

Thirteen (57%) of the 23 software packages use GitHub for managing the project, including nine (64%) of the 14 alive packages, and four of the nine (44%) dead packages. Two projects (Palabos, WaLBerla) use GitLab. This could be indicative of a transition to such software development and version control tools for SCS. Typically there are several simultaneous development branches in these projects.

Documentation was also noted as playing a significant role in the development process, specifically with on-boarding new developers. A goal of documentation is to lower the entry barrier for these new contributors. The documentation provides information on how to get started, orients the user to artifacts and the source code, and explains how the system works, including the so-called simulation engine and interface. The use of document gen-

eration tools is mentioned in the artifacts of 12 software packages, and was noted during interviews with developers. Sphinx and Doxygen are the tools that were mentioned.

Two types of software changes were discussed during interviews with developers. One is feature additions, which arise from a scientific or functional need. The development of such changes was noted to involve more formal discussions within the development team and the addition of them into the main software branch 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 tracking is commonly used to manage these changes.

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

5.4 Pain Points

This subsection answers the research question: What are the pain points for developers working on research software projects? What aspects of the existing processes, methodologies and tools do they consider as potentially needing improvement? How should processes, methodologies and tools be changed to improve software development and software quality?

Developers were asked to comment on obstacles in their development process, obstacles encountered by users, and potential future obstacles.

A developer of pyLBM noted that their small development team has a lack of time to

awkward
sentence

implement new features. Small development teams are common for LBM software packages. Team members are almost always part of the same institute or already know each other from other projects. External contributions are rare despite many of the projects accepting them. Aside from on-boarding new developers, time constraints could be mitigated by increasing developer efficiency, which could be addressed in several ways, including by improving the quality of documentation, or incorporating automatic code generation.

A lack of software development experience was noted by the developer of TCLB. Many of the team members on their project are domain experts and there can be a steep learning curve before these team members contribute good quality source code. It was further noted that this has been somewhat addressed, as the code has been re-written to best ensure ease of future contributions.

The same developer also noted that there is a lack of incentive and funding in academia for developing widely used scientific software. The importance of funding for scientific software has been discussed in [10], which notes that “software tools are developed and maintained only for as long as there is explicit or implicit funding”. The developer further commented that there are no journals that publish such scientific software source code. However, there are ways to get such source code cited. Work has been done to address this in [31], which presents a set of software citation principles and discusses “how they could be used to implement software citation in the scholarly community” [21].

Another raised concern was that there are no organizations helping with the development of good quality software; but some do exist, including [Better Scientific Software \(BSSw\)](#), [Software Sustainability Institute](#), and [Software Carpentry](#). Some SCS developers may not be familiar with these organizations.

Scientific software is often developed in-house by the very researchers that temporarily

use it in their own research. Empirical studies of such “professional end-user development” of SCS is noted in [30]. This kind of software has a defined user and purpose, and often does not meet the standards that would be required by external users. It has been categorized as a “private tool” by [10], which notes that despite often being made freely available, “it is not always clear that it is sufficiently mature in terms of domain coverage, validity, documentation or usability, to be useful to other researchers”. It is less common for such ad hoc scientific software to become “user-ready software”, which “is not only research-ready, but should have most of the attributes commonly expected of commercial software products including broadness of scope, robustness, demonstrable correctness and adequate documentation” [10]. As software becomes user-ready, it can become commercialized and closed-source. Documentation of LBM software is noted as being important, but its quality could be improved. As already noted, there is often no time or funding for maintaining quality documentation for software that is rarely used outside of the development team. Furthermore, the documentation generally only provides a shallow overview of the underlying CFD theory. Users would be well advised to already be familiar with these topics, or they should spend significant time referencing theory resources. The documentation instead generally focuses on explaining how to use the software. It is of course not feasible for package documentation to address the underlying physics topics in detail, so it is advised that the package documentation links to resources that better explain the underlying theory. Sometimes, frequently asked questions about the underlying theory are answered in the documentation. OpenLB has an artifact for such questions.

new
Paragraph

Setting up parallelization was also noted as a technical pain point by one of the developers, and the introduction of continuous integration by another. Software development knowledge, and automatic code generation, could mitigate such pain points. As already

noted, many of the developers are domain experts and not professional software developers. The developer of TCLB noted that eliminating equivalent statements using macros had helped improve the quality of their source code, specifically helping with reusing code to run on both the CPU and GPU.

Difficulties with ensuring correctness were also noted by several developers. They indicated that tests are run on all new source code additions, testing both individual modules and the system to verify correctness. These tests compare the package output to known correct output using test cases. The developer of TCLB commented that the amount of testing data that is needed for some cases is a problem as free testing services do not offer capabilities to store and process such large amounts of data, and in-house testing solutions needed to be created to address this limitation. The solution for this has been to limit the size of the testing problems, and to run tests in small batches with few iterations.

A few obstacles related to users were found. Several developers noted that users sometimes try to use incorrect LBM method combinations to solve their problems. Furthermore, some users think that the packages will work out of the box to solve their cases, while the packages require both a good understanding of CFD and an understanding of the requirements for formulating problems in the individual packages, which can be a significant endeavor. These software packages are not like commercial software packages. They are generally set up to solve specific research problems, and are often primarily used by their developers. While they are modifiable to solve similar problems, these modifications are not trivial. Better documentation, with attention on traceability, and automatic code generation are suggested when designing software for change, and would help with these modifications. So far this problem of on-boarding new users has been addressed by updating the documentation to better inform users of the underlying LBM theory and package

requirements. Similar issues with LBM parameters were noted by another developer. Updating the user interface to better explain theoretical principles, as well as test user input for compatibility, was the implemented solution. As noted above, sometimes frequently asked questions on the underlying theory and on how to use the software are answered in the documentation.

The interviewed developer of ESPResSo commented that parts of their package's source code had been refactored to Python to help address usability issues. Python was perceived as a much more usable language, and it would be easy for future users and developers to learn and understand the source code.

A few potential future obstacles were noted. The developer of ESPResSo noted that their source code had been written with a specific application in mind and that due to this there was too much coupling between components of the source code. This results in technical debt, having an impact on future modifiability and reusability when trying to extend the software, and the code would need to be refactored.

As noted above, difficulties with ensuring future correctness could also arise. As new methods and functionality is added into the software, new test cases and test data will need to be developed.

The developer comments have emphasized an importance on source code, while documentation seems to be of secondary importance. It must be stressed that improving documentation could benefit development and help eliminate some of the developer concerns that were raised. The use of automatic document generation tools that capture scientific and computing knowledge, and transform it into software artifacts, is advised. Drasil is an automatic document generation tool that is further discussed in Section 6.1.

There is repetition b/w sections, but I don't think we need to fix this. It is easier for someone to read if there is a little repetition like this.

5.5 Quality Recommendations

This subsection answers the research question: For research software developers, what specific actions are taken to address software qualities?

The following points regarding software quality should be considered when developing LBM software packages. These points are based on developer interviews, SCS literature, what was found to have worked for packages that were designated as high quality in this assessment.

5.5.1 Installability

- Include OS compatibility, including specific OS versions.
- Provide complete installation instructions.
- Installation instructions should be written as if the user does not have any dependencies installed and is installing on a clean OS. This can be tested on a clean environment using a virtual machine.
- List all dependencies in the installation instructions, dependency versions, and how to install them.
- If possible, automate the installation of dependencies. Use tools such as Make.
- Automate the installation process as much as possible. Use tools such as Make.
- The installation instruction should only be in one location.
- Include descriptive error messages for errors encountered during installation. This was done by several top ranked packages, including Ludwig and LUMA

- Provide a way to validate the installation. This can be done using a custom script, or a test case that specifies expected output. This was done by several top ranked packages, including OpenLB, Ludwig, LUMA, and Palabos.
- Provide instructions for uninstalling the software. These were included with pyLBM.

5.5.2 Surface Correctness and Verifiability

- Use a requirements specification document. A template is presented in [37]. *potential*
- Make public (on package website or GitHub) 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 experienced software developers.
- Use and make public (on package website or GitHub) detailed documentation. Consider using automatic document generation tools like Doxygen, Drasil, or Sphinx.
- Provide detailed tutorials that include expected output, like the [waLBerla tutorials](#). ✓
- Use unit tests during development and make them public (on package website or GitHub). This was available for Ludwig. ✓
- Modularize the source code, separate components, hide information behind well defined interfaces.
- Use continuous integration tools (Bamboo, Jenkins, and Travis CI) and processes during development. This was done by top ranked packages ESPResSo and Ludwig. ✓

5.5.3 Surface Reliability

- Include descriptive error messages where appropriate. This was done by most packages that encountered a fault.
- In case automatic installation of dependencies fails, the system should indicate to the user what dependencies need to be installed manually. This was done by many packages, including ESPResSo++, Ludwig, LUMA, pyLBM, TCLB.
- The packages should include detailed tutorials, including dependencies, expected output, and any additional supplementary documentation that may be required. This was done by many packages, including waLBerla, Palabos, MechSys, LUMA, pyLBM.

5.5.4 Surface Robustness

- The system must provide descriptive error messages when it encounters unexpected input, including incorrect data types, empty input, and missing files or links. Eighteen of the software packages behaved reasonably when tested with unexpected input.

5.5.5 Surface Performance

- Integrate parallelization tools and techniques to reduce processing time. Consider GPU processing, CUDA, and MPI. Seventeen software packages mentioned parallelization in their artifacts.
- The user should be able to choose to process their model on either the CPU or GPU. ESPResSo, lbmpy, lettuce, pyLBM, and TCLB allow for either CPU or GPU processing.

5.5.6 Surface Usability

- Include user hardware and software requirements documentation. Hardware requirements are rarely listed in the software packages that were assessed. Only those offering GPU processing (ESPResSo, lbmpy, lettuce, pyLBM, Sailfish, TCLB) mentioned any hardware requirements. On the other hand, some sort of software requirements were available for all packages, even if only some dependencies or a compatible operating system were mentioned.
- Provide a user tutorial that indicates the expected output. This was done by many packages, including waLBerla, Palabos, MechSys, LUMA, pyLBM.
- Provide a detailed user manual. It should identify elements of user interfaces, and identify all requirements to model a system. Thirteen packages have a user manual. ESPResSo has a well detailed manual.
- State appropriate fluid dynamics problems that the software is designed to model in the documentation, and explicitly state the limits of the software. This is done in the ESPResSo user guide.
- Provide documentation that details the background theory information, or provide a reference to such information.
- Identify expected user characteristics. LIMBES, Ludwig, laboetie, and Palabos did this. The importance of specifying user characteristics is discussed in [\[35\]](#).
- Keep all documentation in one location.
- Do not duplicate artifact information.

- Maintain a user support model (Git, email, forum, FAQ)
- If possible, consider using popular user-friendly software languages like Python. Especially consider this for parts of the source code that is likely to be modified or reviewed by users. ESPResSo and Sailfish use Python to shorten development time and improve usability.

5.5.7 Maintainability

- Keep artifacts updated.
- Ideally include most of the common and less common artifacts listed in Section 5.1.
- Include version numbers and release notes for all major source code and artifact releases. The top five ranked software packages include release notes.
- Have a defined process for accepting contributions, and make public documentation for making contributions to the project. The top five ranked software packages include information on how to contribute.
- Use an issue tracker (Git, email, SourceForge, other) to manage bugs and changes. Issues should be regularly reviewed and closed. All but three (laboetie, lettuce, Sailfish) of the software packages that use Git have most of their issues closed.
- Use a version control system (GitHib, CVS). Four (ESPResSo, Ludwig, LUMA, Palabos) of the top five ranked packages use a version control system.
- Source code needs to be well commented. Typically, more than 10 percent of LBM package source code is comments, as presented in Table 3.

Low correlation with quality of comments?

- Modularize the source code, separate components, hide information behind well defined interfaces.
- Eliminate code duplication.
- If possible, consider using popular user-friendly software languages like Python. Especially consider this for parts of the source code that is likely to be modified or reviewed by users. ESPResSo and Sailfish use Python to address several software qualities, including maintainability.
- Consider the recommended points for addressing traceability.

5.5.8 Modifiability

- Modularize the source code, separate components, hide information behind well defined interfaces.
- If possible, consider using popular user-friendly software languages like Python. Especially consider this for parts of the source code that is likely to be modified or reviewed by users. ESPResSo and Sailfish use Python to address several software qualities, including modifiability.
- Consider future source code modifiability as early as the design stage of development.
- Consider flexibility of data structures and data storage in the design stage. The package pyLBM redeveloped data structures to ease future changes.

5.5.9 Reusability

- Modularize the source code, separate components, hide information behind well defined interfaces.
- Document module interfaces in design and developer documentation.
- Provide API documentation, if applicable. Only one (ESPResSo) of the top five ranked packages provided API documentation. ✓

5.5.10 Surface Understandability

- Adopt a coding standard and document it in artifacts, including some examples. A coding standard needs to be part of continuous integration. Only LUMA and HemeLB explicitly identify coding standards.
- Use consistent, distinctive, and meaningful code identifiers. Nineteen of the 23 software packages use such code identifiers.
- Use symbolic constants. Twelve of the 23 software packages use such constants.
- Identify algorithms that are used in source code comments. Document them in the artifacts. Cite relevant external sources if needed. Domain algorithms are noted in the source code of 11 packages.
- Add meaningful comments. Indicate what is being done in each section of source code.
- Modularize the source code, separate components, hide information behind well defined interfaces.

- Provide a user manual that identifies all technical and problem modeling requirements. Thirteen software packages have a user manual.
- State appropriate fluid dynamics problems that the software is designed to model in the documentation, and explicitly state the limits of the software. This is done in the ESPResSo user guide.
- Consider adding a FAQ section to the documentation. This helped resolve some usability issues for OpenLB.

5.5.11 Traceability

- Update all relevant documentation when a change to the software is made.
- Use automatic document generation tools (Doxygen, Drasil, Sphinx) to limit the time spent on updating documentation.
- Provide a developer's guide to help orient developers to the artifacts, source code, and system architecture so that they can better document changes.

5.5.12 Visibility and Transparency

- Summarize the development process that is used. Provide information on how new users can contribute. Identify the development model by name (waterfall, agile, etc.), if appropriate.
- Identify the development environment. Eight of the 23 software packages identify the development environment.
- Update all relevant documentation when a change to the software is made.

- Include notes with all releases. Nine of the packages include release notes.
- Communicate all changes within the development team.
- Use continuous integration processes and tools (Bamboo, Jenkins, and Travis CI). This was done by top ranked packages ESPResSo and Ludwig.
- Consider peer review processes to assess contributions and ensure the tracking of information. High ranked package ESPResSo uses a peer review process for contributions.
- Use project management tools, including change and version control tools (GitHub, GitLab, CVS), collaboration tools (GitHub, GitLab), and document generation tools (Doxygen, Drasil, Sphinx).

5.5.13 Reproducibility

- Test output against automatically calculated or known correct results. High ranked package ESPResSo does this.
- Automate the testing of output. Run tests after all code changes. High ranked package ESPResSo does this.
- Consider peer review processes and task based inspection to assess contributions. High ranked package ESPResSo does this.
- Update all relevant design documentation when a change to the software is made.

5.5.14 Unambiguity

- Documentation must state all technical requirements and software dependencies. A missing dependency was a frequent cause of fault conditions during this assessment.
- Provide a detailed user manual. Thirteen packages have a user manual. ESPResSo has a well detailed manual.
- Provide a table of symbols in the developer's guide that maps to names used in the course code.
- State appropriate fluid dynamics problems that the software is designed to model in the documentation, and explicitly state the limits of the software. This is done in the ESPResSo user guide.
- The documentation should either explain the underlying CFD theories or provide a reference to appropriate resources.
- Consider asking users for feedback on the documentation. The developer of pyLBM noted that such feedback would be appreciated.

5.6 Designation Comparison

This subsection answers the research question: How does software designated as high quality by this methodology compare with top rated software by the community?

This comparison helps evaluate the methodology that was used, and assess the validity of the findings. Threats to the validity of this exercise are noted in Section 5.7. The software package designations of this report are compared to package repository ranking metrics in Section 5.6.1, and to software recommended by a domain expert in Section 5.6.2.

5.6.1 Repository Ranking Metrics

Excellent!

Name	Our Ranking	Repository Stars	Repository Watches
Ludwig	1	19	6
ESPresSo	2	115	23
Palabos	3	14	GitLab
OpenLB	4	No Git	No Git
LUMA	5	26	10
pyLBM	6	57	8
DL.MESO (LBE)	7	No Git	No Git
waLBerla	8	17	GitLab
Sailfish	9	154	39
laboetie	10	4	5
TCLB	11	61	16
MechSys	12	No Git	No Git
lettuce	13	8	2
ESPresSo++	14	31	13
MP-LABS	15	7	2
SunlightLB	16	No Git	No Git
LB3D	17	No Git	No Git
LIMBES	18	No Git	No Git
LB2D-Prime	19	No Git	No Git
HemeLB	20	22	15
lbmpy	21	4	2
LB3D-Prime	22	No Git	No Git
LatBo.jl	23	4	5

add a column for repository stars rank (it is easier to compare to our rank)

Table 5: Repository Ranking Metrics

Table 5 presents our LBM software package rankings along with the repository ranking metrics of each software package.

Eight packages do not use GitHub or GitLab, so they do not have a measure of repos-

itory stars. Looking at the repository stars of the other 15 packages, we can observe a slight pattern where packages that have been highly ranked by our assessment do have more stars than lower ranked packages. The second ranked package (ESPResSo) has the second most number of stars. The ninth ranked package (Sailfish) has the most number of stars. Packages designated as lower quality often do not use GitHub or GitLab, or have few stars, except for HemeLB which has 22 stars. Our assessment of this package might not be accurate.

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

Besides missing data, another threat to the validity of this comparison is the varying ages of the repositories. Older packages have been able to accumulate stars and watches for longer than newer packages. The true quality of new packages may not be reflected in their stars and watches.

5.6.2 Domain Expert Recommended Software

Table 6 compares the top 10 ranked LBM software packages with a LBM package ranking made by a domain expert. Five (Palabos, OpenLB, LUMA, waLBerla, Sailfish) of the top 10 ranked packages in this assessment are also found in the domain expert's list. Interestingly, these five packages are listed in the same order on both lists. Looking at the remaining packages, the top two ranked packages in this state of the practice assessment (Ludwig, ESPResSo) are not on the domain expert's list. Perhaps the intended applications

- You should include a note that the number of stars does not necessarily represent the perceptions of the community, but for lack of an alternative measure, we will use it for now.

✓ of these packages, complex fluids for Ludwig and soft matter research for ESPResSo, do not align with the research interests of the domain expert, and this is why they did not make it onto their list. Moving down the list, pyLBM and DL_MESO(LBE) did not make the domain expert's top 10 list but were mentioned by the domain expert. The 10th ranked package, laboetie, was not mentioned by the domain expert. Looking at the remainder of the domain expert's list, ASL is a general purpose tool for solving partial differential equations and may have fallen out of scope of the authoritative lists that were used to identify the initial LBM software list. The domain expert's fifth ranked package (ch4-project) was on the initial software list of this assessment, but was removed due to a lack of documentation. Open FSI is a new project that uses Palabos. It was made public the year before data was collected for this state of the practice exercise. It may not have been on authoritative lists due to its age. Similarly, LIFE was also made public recently. Finally, the domain expert's 10th ranked package (HemeLB) was ranked 20th in this state of the practice assessment.

young —————

put the domain expert rankings in brackets

Rank	This Assessment	Domain Expert
1	Ludwig (NA)	Palabos
2	ESPResSo (NA)	OpenLB
3	Palabos (1)	LUMA
4	OpenLB (2)	ASL
5	LUMA (3)	ch4-project
6	pyLBM (NA) etc.	Open FSI
7	DL_MESO (LBE)	WaLBerla
8	waLBerla	LIFE
9	Sailfish	Sailfish
10	laboetie	HemeLB

Table 6: Ranking Comparison

5.7 Threats To Validity

This section examines potential threats to the validity of this state of the practice assessment. These can be categorized into methodology and data collection issues.

The measures listed in our measurement template may not be broad enough to accurately capture some qualities. For example, there are only two measures of surface robustness. The measurement of robustness could be expanded, as it currently only measures unexpected input. Other faults could be introduced, but could require a large investment of time to develop, and might not be a fair measure for all packages. Similarly, reusability is assessed along the number of code files and LOC per file. While this measure is indicative of modularity, it is possible that some packages have many files, with few LOC, but the files do not contain source code that is easily reusable. The files may be poorly formatted, or the source code may be vague and have ambiguous identifiers. Furthermore, the measurement of understandability relies on 10 random source code files. It is possible that the 10 files that were chosen to represent a software package may not be a good representation of the understandability of that package.

Regarding data collection, a risk to the validity of this assessment is missing or incorrect data. Some software package data may not have been measured due to technology issues like broken links. This issue arose with the measurement of Palabos, which had a broken link to its user manual, as noted in Section 5.1.2.

Some pertinent data may not have been specified in public artifacts, or may be obscure within an artifact or web-page. The use of unit testing and continuous integration was mentioned in the artifacts of only two (ESPResSo, Ludwig) packages. However, interviews suggested a more frequent use of both unit testing and continuous integration in the development processes than what was observed from the initial survey of the artifacts. For

Point out
in this
section that
for good
reason so
much software
is software, but to
use the context
exercise as a means
to understand
the state of
the practice.

example, OpenLB, pyLBM, and TCLB use such methods during development despite this not being explicitly clear from an analysis of the material available online.

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

6 Conclusion

We analyzed the state of the practice of software development of Lattice Boltzmann Methods software domain by quantitatively and qualitatively measuring, and comparing, 23 software packages along quality attributes. The software qualities that were assessed in this report are listed in Section 3.2. A methodology for assessing the state of the practice of software development in ~~scientific computing~~^{Sc} software domains was presented. Domain packages were assessed to answer the software development related research questions listed in Section 1.1 to understand how software quality is impacted by software development choices, including principles, processes, and tools. Software developers were interviewed to identify development pain points, and to identify how software quality is ensured. Quantitative data was used to rank the software packages using ~~an analytical hierarchy process~~^{ANP}. The ranking designations were compared with rankings from the software development community, and we found that many of our top 10 ranked packages are ranked highly by a domain expert. Recommendations for improving software along quality metrics were made, and highlights are presented in Section 6.1 of this conclusion. The findings of this report can be used to guide future development of ~~scientific computing~~^{Sc} software, specifically along quality attributes, and to reduce software quality failures.

We understand that software packages vary in goals, developers, funding, and other aspects. Our goal was to highlight software quality successes through an evaluation of the entire domain software family. Furthermore, threats to the validity of the findings are highlighted in Section 5.7.

Recommendations for future state of the practice assessments are made in Section 6.2.

6.1 Highlighted Recommendations

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

- Provide a detailed user manual. It should identify elements of user interfaces, and identify all requirements to model a system.
- State appropriate fluid dynamics problems that the software is designed to model in the documentation, and explicitly state the limits of the software.
- Include detailed tutorials, including dependencies, expected output, and any additional supplementary documentation that may be required.
- Keep all documentation in one location.
- If possible, consider using popular user-friendly software languages like Python. Especially consider this for parts of the source code that is likely to be modified or reviewed by users.
- Modularize the source code, separate components, hide information behind well defined interfaces.
- Include descriptive error messages where appropriate.
- Summarize the development process that is used. Provide information on how new users can contribute.
- Consider peer review processes and task based inspection to assess contributions.

- Use continuous integration tools (Bamboo, Jenkins, and Travis CI) and processes during development.
- Use project management tools, including change and version control tools (GitHub, GitLab, CVS), collaboration tools (GitHub, GitLab), and document generation tools (Doxygen, Drasil, Sphinx).

Ensuring software quality can take significant time. An inability to develop high quality 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 [49].

6.2 Future State Of The Practice Assessments

This section notes recommendations for future state of the practice assessments.

As mentioned in Section 5.7, the measures listed in our measurement template may not be broad enough to accurately capture some qualities. Adding or extending measures is worth considering.

Section 4.2 noted that developers were not asked to comment on the reproducibility of their source code from their requirements specifications and design documentation. Adding this question to the interview guide should be considered in the next iteration of state of the practice assessments. Furthermore, as mentioned in Section 5.7, it was found that some pertinent information was not specified in public artifacts. The use of unit testing and continuous integration by several packages (OpenLB, pyLBM, TCLB) was only discovered

in the future

*- Usability experiments
- Performance benchmarks*

Agree Any ideas on how to do this?

during interviews with developers. Adding further questions to the interview guide regarding the measures that are on the measurement template could reduce instances of incorrect data being collected. This additional interview data could be analyzed and incorporated into the AHP ranking, ensuring quality designations more accurately represent the true quality of the software packages.

A 7 Appendix

A 7.1 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?

A7.2 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 7: 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)

7.3 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 8: 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)

father than A1, A2, ... you can use
Appendix A, B, C, etc. whether is easier
in LaTeX

A.4 Eliminated Software Packages

The table below lists the software packages that were filtered out of the candidate software list.

Name	Reason(s) For Removal
ch4-project	Scope & Usage
CUDA-LBM-simulator	Scope & Age
elbe	Scope & Usage
eLBM	Usage
firesim	Scope & Usage & Age
fvLBM	Scope & Usage & Age
JFlowSim	Scope & Usage & Age
LatticeBoltzmann	Scope & Usage
LBM	Scope & Usage & Age
LBM-Cplusplus	Scope & Usage
LBM-EP	Usage
LBM_MATLAB	Scope & Usage & Age
LBSim	Scope & Usage & Age
listLBM	Scope & Usage & Age
loliverhennigh	Scope & Usage & Age
openLBMflow	Scope & Usage & Age
ParallelLbmCranfield	Usage & Age
PowerFLOW	Usage
ProLB	Usage
Taxila-LBM	Usage & Age
turbulent_lbm_multigpu	Usage & Age
wlb	Scope & Usage & Age

Table 9: Eliminated Software Packages

7.5 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

References

- [1] Iso/iec/ieee international standard - systems and software engineering–vocabulary. *ISO/IEC/IEEE 24765:2017(E)*, pages 1–541, 2017.
- [2] 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.
- [3] Martin Bauer, Sebastian Eibl, Christian Godenschwager, Nils Kohl, Michael Kuron, Christoph Rettinger, Florian Schornbaum, Christoph Schwarzmeier, Dominik Thönnies, Harald Köstler, et al. walberla: A block-structured high-performance framework for multiphysics simulations. *Computers & Mathematics with Applications*, 81:478–501, 2021.
- [4] F. Benureau and N. Rougier. Re-run, Repeat, Reproduce, Reuse, Replicate: Transforming Code into Scientific Contributions. *ArXiv e-prints*, August 2017.
- [5] Prabhu Lal Bhatnagar, Eugene P Gross, and Max Krook. A model for collision processes in gases. i. small amplitude processes in charged and neutral one-component systems. *Physical review*, 94(3):511, 1954.
- [6] Barry W Boehm. *Software engineering: Barry W. Boehm’s lifetime contributions to software development, management, and research*, volume 69. John Wiley & Sons, 2007.
- [7] Shiyi Chen and Gary D Doolen. Lattice boltzmann method for fluid flows. *Annual review of fluid mechanics*, 30(1):329–364, 1998.
- [8] Z Chen, C Shu, LM Yang, X Zhao, and NY Liu. Phase-field-simplified lattice boltzmann method for modeling solid-liquid phase change. *Physical Review E*, 103(2):023308, 2021.
- [9] Davood Domairry Ganji and Sayyid Habibollah Hashemi Kachapi. *Application of nonlinear systems in nanomechanics and nanofluids: analytical methods and applications*. William Andrew, 2015.
- [10] Marc-Oliver Gewaltig and Robert Cannon. Quality and sustainability of software tools in neuroscience. *Cornell University Library*, pages 1–20, 2012.
- [11] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of software engineering*. Prentice Hall PTR, 1991.

- [12] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 2003.
- [13] Benjamin Graille and Loïc Gouarin. pylbm documentation. 2017.
- [14] 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.
- [15] 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.
- [16] IEEE. Ieee standard glossary of software engineering terminology. Standard, IEEE, 1991.
- [17] IEEE. Recommended practice for software requirements specifications. *IEEE Std 830-1998*, pages 1–40, October 1998.
- [18] ISO/IEC. *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC, 2001.
- [19] 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.
- [20] Panagiotis Kalagiakos. The non-technical factors of reusability. In *Proceedings of the 29th Conference on EUROMICRO*, page 124. IEEE Computer Society, 2003.
- [21] Daniel S Katz, Daina Bouquin, Neil P Chue Hong, Jessica Hausman, Catherine Jones, Daniel Chivvis, Tim Clark, Mercè Crosas, Stephan Druskat, Martin Fenner, et al. Software citation implementation challenges. *arXiv preprint arXiv:1905.08674*, 2019.
- [22] 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.
- [23] 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.

- [24] 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.
- [25] J. McCall, P. Richards, and G. Walters. *Factors in Software Quality*. NTIS AD-A049-014, 015, 055, November 1977.
- [26] JD Musa, Anthony Iannino, and Kazuhira Okumoto. Software reliability: prediction and application, 1987.
- [27] Jakob Nielsen. Usability 101: Introduction to usability, 2012.
- [28] David Lorge Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, (1):1–9, 1976.
- [29] Mariusz Rutkowski, Wojciech Gryglas, Jacek Szumbariski, Christopher Leonardi, and Łukasz Łaniewski-WoŃk. Open-loop optimal control of a flapping wing using an adjoint lattice boltzmann method. *Computers & Mathematics with Applications*, 79(12):3547–3569, 2020.
- [30] Judith Segal. End-user software engineering and professional end-user developers. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.
- [31] Arfon M Smith, Daniel S Katz, and Kyle E Niemeyer. Software citation principles. *PeerJ Computer Science*, 2:e86, 2016.
- [32] 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.
- [33] Spencer Smith, Jacques Carette, Olu Owojaiye, Peter Michalski, and Ao Dong. Methodology for assessing the state of the practice for domain x.
- [34] Spencer Smith, Jacques Carette, Olu Owojaiye, Peter Michalski, and Ao Dong. Quality definitions of qualities.
- [35] Spencer Smith, Lei Lai, and Ridha Khedri. Requirements analysis for engineering computation: A systematic approach for improving reliability. *Reliable Computing*, 13(1):83–107, 2007.
- [36] Spencer Smith, Yue Sun, and Jacques Carette. State of the practice for developing oceanographic software. *McMaster University, Department of Computing and Software*, 2015.

- [37] W Spencer Smith and Lei Lai. A new requirements template for scientific computing. In *Proceedings of the First International Workshop on Situational Requirements Engineering Processes—Methods, Techniques and Tools to Support Situation-Specific Requirements Engineering Processes, SREP*, volume 5, pages 107–121. Citeseer, 2005.
- [38] W. Spencer Smith, Adam Lazzarato, and Jacques Carette. State of practice for mesh generation software. *Advances in Engineering Software*, 100:53–71, October 2016.
- [39] W. Spencer Smith, Adam Lazzarato, and Jacques Carette. State of the practice for GIS software. <https://arxiv.org/abs/1802.03422>, February 2018.
- [40] 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>, 2018. 33 pp.
- [41] W. Spencer Smith, Zheng Zeng, and Jacques Carette. Seismology software: State of the practice. *Journal of Seismology*, 22(3):755–788, May 2018.
- [42] Ian Sommerville. *Software Engineering 9*. Pearson Education, 2011.
- [43] Richard H Thayer and Merlin Dorfman. Ieee recommended practice for software requirements specifications. *IEEE Computer Society, Washington, DC, USA, 2nd ed. edition*, 2000.
- [44] Nils Thürey and Ulrich Rüde. Stable free surface flows with the lattice boltzmann method on adaptively coarsened grids. *Computing and Visualization in Science*, 12(5):247–263, 2009.
- [45] Axel Van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Proceedings fifth ieee international symposium on requirements engineering*, pages 249–262. IEEE, 2001.
- [46] 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.
- [47] David M. Weiss. Defining families: The commonality analysis. *Submitted to IEEE Transactions on Software Engineering*, 1997.
- [48] Wiegers. *Software Requirements, 2e*. Microsoft Press, 2003.
- [49] Yuzhi Zhao. Automated knowledge extraction based on a scientific computing software documentation generation framework, 2018.