Received: 15 April 2020 | Revised: 15 October 2020 | Accepted: 19 October 2020

DOI: 10.1002/spe.2926

SPECIAL ISSUE PAPER

WILEY

Unit of measurement libraries, their popularity and suitability

Steve McKeever | **Oscar Bennich-Björkman** | **Omar-Alfred Salah**

Department of Informatics and Media, Uppsala University, Uppsala, Sweden

Correspondence
Steve McKeever, Department of Informatics and Media, Uppsala University, Uppsala, Sweden.
Email: steve.mckeever@im.uu.se

Abstract

In scientific applications, physical quantities, and units of measurement are used regularly. If the inherent incompatibility between these units is not handled properly it can lead to potentially catastrophic problems. Although the risk of a miscalculation is high and the cost equally so, almost none of the major programming languages has support for physical quantities. We employed a systematic approach to examine and analyse available units of measurement (UoM) libraries. The search results were condensed into 38 libraries. These were the most comprehensive and well-developed, open-source libraries, chosen from approximately 3700 search results across seven repository hosting sites. Most libraries are implemented in a similar manner, but with varying features and evaluation strategies. Three developers and a scientist were interviewed and 91 practitioners of varying experiences from on-line forums were surveyed to explain their impressions of UoM libraries and their suitability. Our findings show several reasons for nonadoption, including insufficient awareness of UoM libraries, cumbersome in practice, specific performance concerns, and usage of development processes that exclude unit information We conclude with recommendations to UoM library creators derived from these observations. We also argue that so long as units are not part of the language, or not supported through an IDE extension, their use will be limited. Native language support allows for efficient unit conversion and static checking. While lightweight methods provide many benefits of UoM libraries with minimal overheads. Libraries are perhaps best suited to applications in which unit of measurement checking is desirable at run-time.

KEYWORDS

quantity pattern, units checking, units libraries, units of measurement, units survey

1 | INTRODUCTION

On the morning of September 23rd, 1999, NASA lost contact with the Mars Climate Orbiter, a space probe sent up a year prior with the mission to survey Mars. The probe had malfunctioned, causing it to disintegrate in the upper atmosphere.

A later investigation found that the root cause of the crash was the incorrect usage of Imperial units in the probe's software.[1] This seemingly trivial mistake ended up costing more than 300 million dollars and years of work. There are many other such examples where unit mistakes have been catastrophic and very costly. One way of preventing these errors is through the use of unit of measurement (UoM) libraries.

Unit errors can be both mistakes made within one unit system, as in the case of Discovery STS-18 which accidentally ended up being positioned upside down because the engineers had mistaken feet for miles;[3] or errors stemming from wrongful conversions between different unit systems, such as the Mars Climate Orbiter.

To better understand the underlying problem, consider the following scenario. There are two programmers working on a system that manages physical quantities using a popular language such as C#, Java, or Python. The first programmer wants to create two quantities that will be used by the second programmer at a later stage. Because the language does not have support for this type of construct, he or she decides to do it using integers and adding comments, like this:

```
int mass = 10; // in tonnes
int acceleration = 10; // in m/s
```

Now the second programmer wants to use these values to calculate force using the well-known equation `F = m * a`:

```
int force = mass * acceleration; // 100 N
```

The variable `force` will now have the value of `100`, assumed to be `100 N`. The issue is that the variable mass is actually representing `10 tonnes`, not `10 kg`. This means that the actual value of the force should be `100,000 N` (instead of `100 N`), off by a factor of one thousand. Because the quantities in this example are represented using integers there is no way for the compiler to know this information and therefore it is up to the programmers themselves to keep track of it.

The problem could have been avoided if the second programmer had seen the comments and wrote code accordingly. This is in essence the fundamental problem that can lead to the catastrophic events that were described previously. A robust solution is to automate UoM checking to ensure the calculations yield the correct unit. This type of systematic check is potentially something that could be undertaken at compile-time in a strongly typed language, but unfortunately very few languages have support for units of measurement (see Section 2.3). Instead, it is up to the software developers to create these checks themselves.

One example of how this can be achieved is to make sure the compiler knows what quantities are being used by encapsulating this into a class hierarchy, with each unit having its own class. Assuming operator overloading, the scenario above would then look like this instead:

```
Tonne mass = 10;
Acceleration acceleration = 10; // in m/s
Force force = mass * acceleration; // 100,000 N
```

Compared with the previous example, here the compiler now knows exactly what it is dealing with and thus the information that the mass is in tonnes is kept intact and the correct force can be calculated in the end. This type of solution not only means that differences in magnitude and simple conversions are taken care of but also that any erroneous units being used in an equation can be caught at compile-time. An example of this can be seen in Figure 1. In the example shown, the programmer makes a mistake when trying to calculate speed. Instead of using the correct units (which is length over time), he or she uses meters divided by seconds squared. Luckily, the static checker catches this error and informs the programmer.

Making a class hierarchy similar to the one illustrated above could potentially involve hundreds of units and thousands of conversions. One way to tackle this issue and be able to safely use physical quantities without spending precious development resources is to use what others have already done, in the form of free, open-source, software libraries. These libraries can provide the class-structure and logic that is needed and can (often) be integrated into an already existing code base. These types of libraries do already exist, several hundreds in fact. The problem thus is not the lack of these solutions but the opposite that they are so numerous that it is hard to get an overview. Moreover, when looking deeper, it quickly becomes apparent that there is no archetype for UoM libraries and a general lack of cooperation. This results in most library developers creating their own versions from scratch without referencing the work of others. Even though they all try to solve the same fundamental problem, the libraries are developed in isolation and typically "reinvent the wheel."

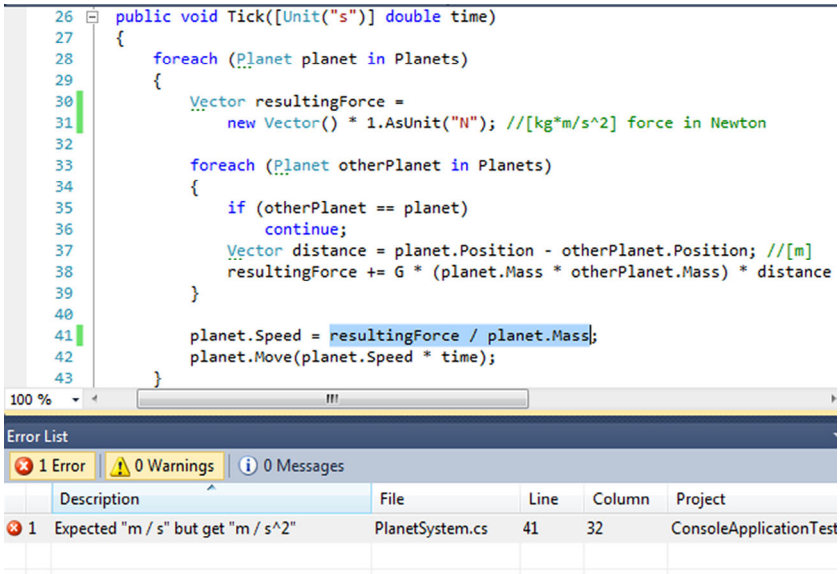


FIGURE 1 Compilation error example from units of measure validator for C#[2] [Colour figure can be viewed at wileyonlinelibrary.com]

The goal of our work is to understand how best to support units of measurement checking in scientific applications. Moreover, we have focused on UoM and not on unit uncertainty as this is less clearly defined in the literature[4] We began with an overview of popular libraries (specifically open-source) that handle the use of physical quantities through a comprehensive and systematic approach. We shall then briefly delve deeper into how these libraries function, noting that they all adopt the same design pattern. Finally, we present the result of a survey into the reasons for nonadoption of UoM libraries. This sheds light on a number of issues that concern developers and leads us to conclude how best to use both UoM libraries and alternative solutions.

The article is structured as follows. In Section 2, we describe UoM systems, we briefly describe the history of unit systems and other surveys that cover unit libraries. The bulk of our UoM library study is presented in Section 3 where we outline the filtering process applied to open-source projects to uncover the state-of-play. We also briefly describe how the most popular UoM libraries are implemented, highlighting similarities and differences. In Section 4, we present our survey of potential users of UoM libraries and their reasons for lack of adoption. In Section 5, we discuss alternative approaches based on component annotations that mitigate some of the downsides to library support. Finally, in Section 6 we consider the results of our survey and provide an overall summary.

2 | BACKGROUND

The technical definition of a physical quantity is a "property of a phenomenon, body, or substance, where the property has a magnitude that can be expressed as a number and a reference".[5]

To explain this further, each quantity is expressed as a number (the magnitude of the quantity) with an associated unit.[6] For example, you could express the physical quantity of length with the unit meter and the magnitude `10` (`10 m`). However, the same length can also be expressed using other units such as centimeters or kilometers, at the same time changing the magnitude (`1000 cm` or `0.01 km`). Keeping the same physical quantity consistent across multiple units is one of the main functions that the libraries presented in this article provides, and something that if not kept in check can have major consequences (as described previously).

Physical quantities also come in two types, one called *base quantities* and the other *derived quantities*. The base quantities are the basic building blocks and the derived quantities are built from these. The base quantities and derived quantities together form a way of describing any part of the physical world.[7] For example, length (meters) is a base quantity, and so is time (s). If you combine these two base quantities you can express velocity (m/s or $m \times s^{-1}$) which is a derived quantity. The International System of Units (SI) defines seven base quantities (length, mass, time, electric current, thermodynamic temperature, amount of substance, and luminous intensity) as well as a corresponding unit for each quantity.[8] These base units were chosen for historical reasons, and were, by convention, regarded as dimensionally independent. As of the May 20th, 2019 all SI units are now defined in terms of constants that describe the natural world, assuring the future stability

Name	Symbol	Quantity	Base Units
hertz	Hz	*frequency*	sec^{-1}
newton	N	*force, weight*	$\mathsf{kg} \times \mathsf{meter} \times \mathsf{sec}^{-2}$
pascal	Pa	*pressure, stress*	$\mathsf{kg} \times \mathsf{meter}^{-1} \times \mathsf{sec}^{-2}$
joule	J	*energy, work, heat*	$\mathsf{kg} \times \mathsf{meter}^{2} \times \mathsf{s}^{-2}$
watt	W	*power, radiant flux*	$\mathsf{kg} \times \mathsf{meter}^{2} \times \mathsf{sec}^{-3}$
square metre	m^2	*area*	meter^{2}
cubic metre	m^3	*volume*	meter^{3}
metre per second	m/s	*speed, velocity*	$\mathsf{meter} \times \mathsf{sec}^{-1}$
cubic metre per second	m^3/s	*volumetric flow*	$\mathsf{meter}^{3} \times \mathsf{sec}^{-1}$

FIGURE 2 Some standard derived units and quantities

of the SI and open the opportunity for the use of new technologies, including quantum technologies, to implement the definitions.[6]

These physical quantities are also organized in a system of dimensions, each quantity representing a physical dimension with a corresponding symbol (`L` for length, `M` for mass, `T` for time, and so forth).

```
type base = L | M | T ...
```

Any derived quantity can be defined by a combination of one or several base quantities raised to a certain power. These are called *dimensional exponents*.[9]

```
type derived = Base of base
             | Times of (derived * derived)
             | Exp of (derived * int)
```

Dimensional exponents are not a type of unit or quantity in themselves but rather another way to describe an already existing quantity in an abstract way. Using the same example of velocity as before, it can be expressed as:

```
Times (L, Exp (T,-1))
```

Alternatively in concrete syntax as $\mathtt{L} \times \mathtt{T}^{-1}$, where `L` represents length and $\mathtt{T}^{-1}$ represents the length being divided by a certain time. Although from a physical perspective each unit can be defined in this way, it is not necessarily the way they are defined in a physical quantity library. The definition allows an infinite number of derived unit types to be created whereas there are a limited number in our physical universe, some shown in Figure 2, so many libraries hard code their allowable units. Therefore, the inclusion or exclusion of dimensional exponents is one potential way of categorizing these libraries.

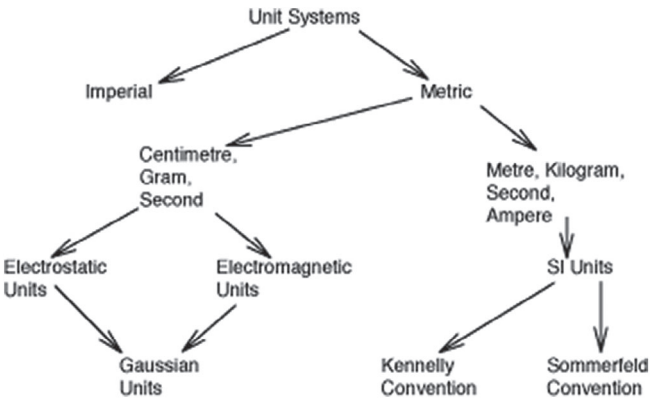
As was previously discussed, performing calculations in relation to quantities, units, and dimensions is often complex and can easily lead to mistakes. One way to try to deal with this (outside of unit libraries) is to do what's called a *dimensional analysis*. One example of a type of dimensional analysis is to check for *dimensional homogeneity*, meaning that both sides of an equation has equal dimensions (which they should have).[7] The concept of dimensional analysis is also relevant to the libraries described in this article, as they often employ similar ways to check for errors. There are also specific software tools that can perform dimensional analysis on existing code (e.g., described by Cmelik and Gehani[10]).

The examples above were all based on the SI which is the most used and well-known unit system, but there exists several other systems that these physical quantities can be expressed in, each with different units for the same quantity. Other examples include the *Imperial system*, the *Atomic Units system*, and the *CGS* (centimeter, gram, second) system, see Figure 3. These have evolved over time and branched off from each other.

To give an example of how these systems relate to units and quantities you can think of how to express a certain length in the SI system and the Imperial system, for example, 2 m is about 6.6 feet. The same quantity (length) is expressed in both instances but by different units, meters in one system and feet in the other.

Although the SI system is the most well used, the Imperial system is still employed heavily in the United States, which means that keeping track of quantities expressed in different unit systems is important and a type of functionality that most quantity libraries implement. Moreover, there are commonly used non-SI units that often need supporting, such as calories or minutes. Allowing user-defined UoM and relevant conversion functions is often a requirement of UoM systems. Moreover, some of these conversion factors might only be known at run-time, for instance, currency conversions.

FIGURE 3 Evolution of units systems



2.1 | The quantity pattern

A more abstract means of representing unit entities is to bind the value along with the unit in what is known as the Quantity pattern.[11]

```
class Quantity {
 private float value;
 private Unit unit;
 ....
}
```

We can include arithmetic operations to the `Quantity` class that ensures addition and subtraction only succeed when their units are equivalent, or multiplication and division generate a new unit that represents the derived value correctly. Moreover, we can include other useful behavior such as printing and parsing to this class.

Representing units using the `base` and `derived` types is not always optimal as a normal form exists which makes storage and, more importantly, comparison a lot easier. Any system of units can be derived from the base units as a product of powers of those base units: $\texttt{base}^{e_1} \times \texttt{base}^{e_2} \times \ldots \texttt{base}^{e_n}$, where the exponents $e_1, \ldots, e_n$ are rational numbers. Thus, an SI unit can be represented as a 7-tuple $\langle e_1, \ldots, e_7 \rangle$ where e_i denotes the ith base unit; or in our case e_1 denotes length, e_2 mass, e_3 time, and so forth. Consequently 3 N would be represented as $\langle 1, 1, -2, 0, 0, 0, 0 \rangle$, or $3\ \texttt{kg.m.s}^{-2}$. Dimensionless units are represented by a tuple whose seven components are all 0. Interestingly any unit from any other system can be expressed in terms of SI units. Conversions can be undertaken using mostly multiplication factors, but in some case offsets are required too. This suggests implementing units through the following class outline:

```
class Unit {
 private int [7] dimension
 private float [7] conversionFactor
 private int [7] offset
 private String name
 ...
 boolean isCompatibleWith (Unit u)
 boolean equals (Unit u)
 Unit multiplyUnits (Unit u)
 Unit divideUnits (Unit u)
}
```

The `dimension` array contains the 7-tuple of base unit exponentials. The attributes `conversionFactor`and `offset` enable conversions from this unit system to the SI units, while `name` is so that users can define their own unit system.

The class `Unit` also defines operations to compare and combine units. The method `isCompatibleWith`checks whether two units are compatible for being combined, such as miles and centimeters. While `equals` returns true if the units are exactly the same, which is used when adding or subtracting quantities. When two quantities are multiplied then

`multiplyUnits` adds the two `dimension`arrays. Correspondingly, `divideUnits` subtracts each of the elements of the `dimension`array.

The idea behind a software design pattern is a general, reusable solution to a commonly occurring problem. The Quantity pattern therefore provides a means of annotating variable declarations and method signatures to specify their UoMs and capture behavioral correctness. Bear in mind though the annotation burden, Ore et al.[12] found subjects choose a correct UoM annotation only 51% of the time and take an average of 136 seconds to make a single correct annotation. The Quantity pattern is still too coarse for most applications and requires further code to make it effective. As there is no agreed interface to the Quantity pattern, noncompatible domain specific instantiations have proliferated.

2.2 | Existing language support

One of the main reasons why third-party libraries are needed to solve the issues related to units of measurement is because there are almost no contemporary programming languages where this exists as a construct. In fact, the only language in the 20 most popular programming languages on the TIOBE index[13] that support units of measurement is Apple's Swift language.[14] The only other well-known language to support units of measurement at the moment is F#.[15] Although compared with Swift, F# is far less popular.

Both Swift and F# have in common that they are relatively recently developed programming languages, Swift in 2014 and F# in 2005. Adding unit checking to conventional imperative, object-oriented and even functional languages using syntactic sugaring is beyond the algorithmic scope of their underlying type checkers. The key feature of units is that the equations that are generated cannot be solved using the Hindley–Milner[16] type system; the correct theory to use is that of Abelian groups.[17]

There has also been a few attempts to provide standard library support for UoM in the Java language through the Java Community(SM) Program[18] and the UOMo project.[19] It is also possible to make the argument that C++ has support for physical quantities through the Boost::Units library which is one of many libraries that Boost.org provides for the language.[20] These libraries are all well made, well documented, and viewed by many as a de facto part of the C++ language. However, our survey of users was less complementary on the practical handling of units in this library. Finally, there are some examples of smaller languages that support physical quantities, such as Nemerle[21] and Frink.[22]

2.3 | Related work

Adding units to conventional programming languages has a rich history going back to the 1970s[23] and early 1980s with proposals to extend Fortran[24] and then Pascal.[25] However, these efforts were heavily syntax based and required modifications to the underlying languages. Moreover, the connection with standard static checking was not understood.

The pioneering foundational work was undertaken by Wand and O'Keefe[26] in which they show how to add dimensions to the simply typed lambda calculus, such that polymorphic dimensions can be inferred in a way that is a natural extension of Milner's polymorphic type inference algorithm. Andrew J. Kennedy extended Wand's work[17] and contributed greatly to the F# project.[27] However, this strand of work relies on extending the language's static checker. Antoniu et al.[28] developed a tool for checking Excel spreadsheets along similar lines.

A more viable starting point for our study would be the work of Hilfinger[29] that showed how to exploit Ada's abstraction facilities, namely, operator overloading and type parameterization, to assign attributes for units of measure to variables and values. A strength of libraries is that they can support both compile-time and run-time error detection. We show in Section 4.3 how to define a UoM using both overloading and overriding in Java for the unit of beauty Helen. A weakness of libraries compared with native language support is that variables of a Quantity class can be reassigned at run-time so that a metercould become a kilogram.

An alternative pathway is to introduce physical units into an object-oriented modeling platform, along with a compilation workflow that leverages OCL expressions[30] or staged computation[31] to derive units where possible at compile-time. Similarly Gibson et al.[32] add units to the Event-B modeling language and leverage the Rodin theorem prover to detect inconsistencies before translation to Java. These elegant abstractions lift the declaration and management of units into software models. However, once the code has been generated, UoM information might very well be lost unless the workflow has been tailored explicitly.

The largest academic contribution to describing unit library usage was a talk given by Trevor Bekolay from the University of Waterloo at the 2013 SciPy conference.[33] In his 20 min presentation, entitled "A comprehensive look at representing physical quantities in Python," Bekolay discusses why he thinks this type of functionality is essential for any language which sees heavy use in scientific applications (such as Python). He compares and contrasts 12 existing Python libraries that handle this, going through their functionality, syntax, implementation, and performance. In the end of the talk he also presents a possible "unification of the existing packages that enables a majority of the use cases."

There has been some attempts at trying to summarize libraries in this area by practitioners, namely, developers working on their own tools. Most of these are often limited by the author only looking at libraries in a single language and contain little detail. These lists are mostly found in the documentation of physical quantity libraries hosted on GitHub. There are also a few places where one can find collections that go beyond simply listing libraries in one language.[34-37] This motivated us to perform our initial study of unit libraries,[38] in order to get a bird's eye view of the field, and our second[39] to look further into how they're implemented. Section 3 contains a summary of these two articles in order to motivate our user survey.

Unfortunately we lack an authoritative estimate of how frequently unit inconsistencies occur or their cost. Anecdotally we can glean that it is not negligible from experiments described in certain articles. Cooper and McKeever[40] developed a validation tool for CellML, a domain specific language for modeling biological systems. He applied it to the repository of CellML models and, of those that were found to be invalid, 60% had dimensionally inconsistent units. Antoniu et al.[28] applied their spreadsheet checker to 22 published scientific spreadsheets and detected three with errors. Similarly, the Osprey type system[41] provides an advanced unit checker and inference engine for C. In their article they describe having applied it to mature scientific application code and found hitherto unknown errors. Unfortunately they do not describe the prevalence or magnitude of these errors. A more telling statistic is found in Ore[42] where they apply their lightweight C++ unit inconsistency tool to 213 open-source systems, finding inconsistencies in 11% of them. A further study[43] using a corpus of robot software with 5.9M lines of code, found dimensional inconsistencies in 6% of repositories.

3 | LIBRARY STUDY

The first part of our study was spent finding, summarizing, and categorizing similar libraries found on open-source hosting sites during the spring and summer of 2018.[38] As there was almost no previous research in this area, the manner in which this was carried out could not be copied from other researchers but instead had to be designed for this article with this specific goal in mind.

3.1 | General search

The first objective in the library search process was to construct a list of keywords that could be used when searching for projects. Considering what was easily uncovered on Google and GitHub, a list of 10–15 relevant projects were initially found. Based on how the authors described these projects, eight keywords were chosen to describe the general area. These keywords were either used in the title of the projects that were found, in tags for these or in the documentation. The chosen keywords were: "Units of Measure," "Units Measure Converter," "Units," "Unit Converter," "Quantities," "Conversion," "Unit Conversion," and "Physical Units."

After this, seven open-source repository hosting sites were chosen. The intention was to use the eight keywords described above on these seven sites to find as many projects as possible. These sites were chosen based on relevance and popularity.[44-46] The sites were: GitHub.com, BitBucket.org, GitLab.com, SourceForge.net, CodeProject.com, LaunchPad.net and Savannah.gnu.org.

This combination of sites and keywords returned over 65,000 total (nonunique) projects, 78% of these being hosted on GitHub. Most of these results came from just two of the keywords on just one of the sites (GitHub), namely, "Units" and "Conversion." "Units" returning over 30,000 results and "Conversion" over 17,000. This is because they are the most general keywords in the list and because in relation to software projects, "Units" and "Conversion" will relate to several things that are not relevant to the results of this article (such as conversion of data types or unit testing).

As the total number of projects was too large to go through individually, a choice was made to put a cap of a maximum of 200 results (circa 20 pages) per keyword per site. Although the exact number is largely chosen arbitrarily, 200 results was seen as a good compromise between thoroughness and feasibility. The relevance of projects outside of the first 200

TABLE 1 Keyword search results

Search term	GitHub	BitBucket	GitLab	SourceForge	CodeProject	LaunchPad	Savannah GNU
Units of measure	200 (210)	15	1	17	97	12	0
Units measure converter	24	2	0	54 0	2	0	0
Units	200 (31,019)	200 (1673)	69	200 (738)	200 (4979)	75^{max} (254)	4
Unit converter	200 (1369)	55	13	48	16	11	0
Quantities	200 (1586)	88	6	87	200 (772)	29	1
Conversion	200 (17,110)	200 (778)	200 (222)	200 (3816)	200 (3087)	75^{max} (200)	10
Unit conversion	200 (822)	25	12	56	26	4	0
Physical units	125	15	0	30	19	8	1
Sum	1349 (52,085)	600 (2651)	32 (323)	692 (4846)	758 (8980)	216 (520)	16 (16)

results would be low and therefore the chance of missing an important contribution would be low. This cap decreased the number of projects from 65,000 down to about 3700, which was the final number of results that were checked and from which the most comprehensive libraries were chosen.

All the sites and search terms are listed in Table 1 and the number in each cell represents the total number of search results for that combination. If a cell has a number in parenthesis this means that the cap of 200 was employed and the number in the parenthesis was the total number of search results that would otherwise be available. For example, 200 (17,110) for the keyword "Conversion" on the site GitHub means that 200 results for this keyword on this site were looked at but 17,110 was the total amount. In the case of the site "LaunchPad.net," the search function only showed the first 75 results. To indicate where this has affected the amount of search results 75^{max} has been written before the total number of results that would otherwise be available, is shown.

3.2 | Analyzing project validity

After the initial search, all the relevant projects that were found were analyzed again. During this second scrutiny, some of these projects that initially looked relevant were deemed to be invalid and were therefore excluded. Due to the overwhelming popularity of the English language online and in particular for computing related activities, we precluded projects that did not have adequate English documentation.

A project was deemed invalid if at closer inspection it was lacking in one or several areas. These were for example that *the code was incomplete*, *the documentation was severely lacking*, *the code did not work*, or *that the solution was too limited in scope*. In addition, if the source code could not be accessed for whatever reason the project was not included as this was a requirement. Similarly those that were defined as *tools* rather than *libraries* were filtered out. This filtering was undertaken by the second author and checked by the first.

3.3 | Categorizing top tier and second tier libraries

In this final step, the libraries that had been filtered out in the previous step were divided into two groups which were labeled *top tier* and *second tier*. As the name implies, these two groups reflect the quality of the libraries placed in each group, the top tier group being the best examples of physical quantity libraries. As the goal in this part of the process was to categorize the libraries in terms of overall quality (without having to look through every line of code), we attempted to triangulate "quality" based on several factors of the library projects, namely:

- **The library was actively being worked on or had recently been updated**—we took this to mean that developers were improving performance and responding to users requests.
- **The library had a high number of commits**—similarly, we took this to mean that bugs were being addressed and functionality added. Some libraries had as few as 5–10 commits while others had hundreds or even thousands. For the purposes of this article, a high number of commits was seen as anything above 100.

- **The library had a high number of contributors**—many contributors is a viable indirect measurement of quality as it implies diverse people have worked on the project and contributed. Even though this is not a perfect measure of quality either, a project with 50 contributors will generally be more well developed than a project with one contributor.
- **The library had comprehensive documentation**—as we had to look at an extensive list of libraries written in many different programming languages we often had to rely on the documentation that was provided by the developers themselves, rather than being able to understand everything based on the code alone. This meant that the quality and breadth of the information became an important factor. Good documentation is also important to attract and inform potential users.
- **The library supported many different units and unit systems**—the ability for the library to support many different units and unit systems is a fundamental part of what these libraries are used for, and therefore a library that supports more units can generally be said to be of higher quality.
- **The library had high ratings and was popular**—high ratings is usually a good indication of overall quality. Similarly, a library that many people like and is popular is generally of good quality.

If a library was missing one or several of the above criteria, it was instead placed in the second tier group.

3.4 | Results

The overall search results included 3700 projects. From these, 586 projects were found to be relevant. Out of the 586 relevant projects, 391 were valid and looked at further. From the group of 391 valid projects, 296 were libraries and 95 were tools. From the group of 296 libraries, 214 were deemed to be second tier and 82 to be top tier as shown in Figure 4. Of the 82 top tier projects, 38 had been updated in the 2 years prior to the original study.[38]

For the top tier group the total number of commits was about 36,000, an average of about 440 commits per project. The average number of contributors was circa 8.3 per project. However, these numbers were heavily influenced by the Astropy[47] project which had almost 22,000 commits and 240 contributors alone, making it an extreme outlier in the group. Removing the Astropy data, the total number of commits drops to 14,000 and the average commits per project to 175. The average number of contributors per project also goes down to ~5.5.

Seventy-five percent of the projects were updated within the last 2 years (2017 and 2018) of the study, and many projects are being actively worked on and updated continuously. Looking at UoM support, on average, each library supports about 200 unique units. Regarding programming language distribution in this group, there are 30 different languages in total and C# being the most popular, closely followed by Python and Ruby (see Figure 5). It is obvious why compiled languages suitable for numerical computations, like C and C++, would benefit from unit checking; and why Python which is seen as

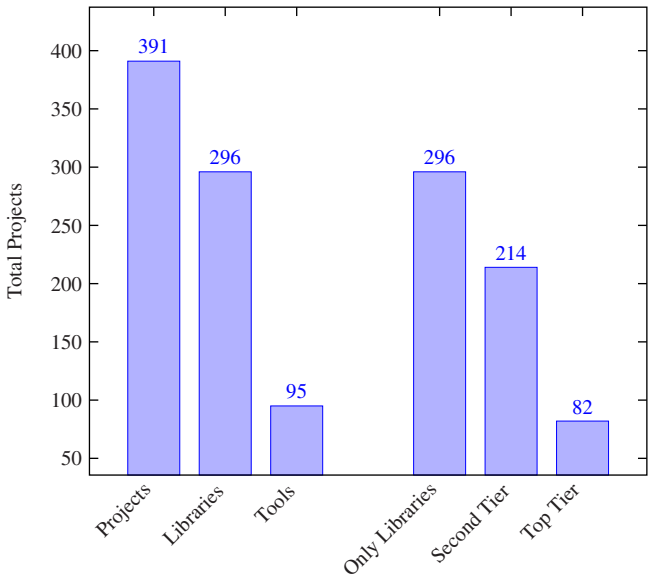


FIGURE 4 Composition of valid projects into tools and libraries, and then into top and second tier libraries [Colour figure can be viewed at wileyonlinelibrary.com]

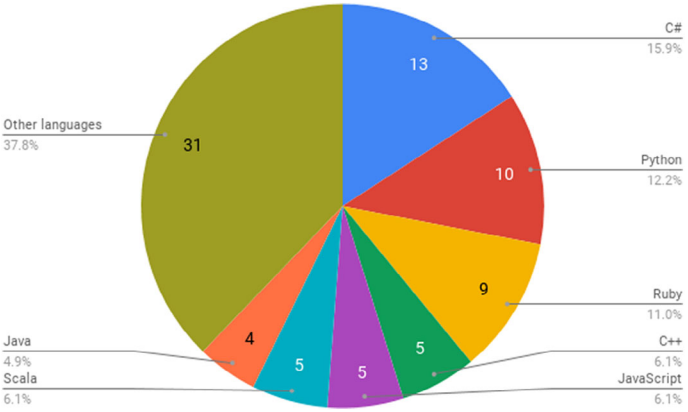


FIGURE 5 Libraries per language [Colour figure can be viewed at wileyonlinelibrary.com]

a scientific scripting language would too. However, it is less clear why libraries for Ruby would be so popular. Whether it is because of its typical application area, or its suitability for creating internal DSLs with a lightweight syntax and dynamic but strongly typed system was not ascertained in either the interviews or survey of Section 4.

Another possible reason for the proliferation may be attributed to the Logic of Collective Action.[48] Its central argument is that concentrated minor interests will be overrepresented and diffuse majority interests due to a *free-rider problem* that is stronger when a group becomes larger. This is perhaps an explanation as to why some of the libraries have continued to exist and prosper when there are equally good ones for that particular language. Once developed and a user base has accrued, due to the open-source nature of the endeavor, it becomes necessary to keep supporting the library as vital code has become dependent on its existence.

3.5 | Observations of library study

A second part of our study looked in more detail at a number of these prominent libraries to elucidate how they operate, their feature set and potential for interoperability.[39] All of the libraries we analyzed implemented the Quantity pattern, albeit in a number of different ways. Some examples of the different implementations are given below:

- **Java**: CaliperSharp(`github.com/point85/CaliperSharp`) implements the Quantity pattern through the use of the class `Quantity`; while JSR 385 (`github.com/unitsofmeasurement/unit-api`) implements the Quantity pattern through the generic interface `Quantity<T>`.
- **C++**: BoostUnits (`github.com/boostorg/units`) is the leading library for C++. It is in effect the de facto standard and is actively developed, has very good documentation, supports many units as well as numerous different constants. BoostUnits is also part of a big development team (`Boost.org`). However, Boost exploits the C++ template meta-programming library so it is more than just a library as it supports a staged computation model similar to MixGen.[31] It implements dimensional analysis in a general and extensible manner, treating it as a generic compile-time meta-programming problem, this style of software development is radically different than a single-stage compilation and is similar to a language with units built-in. The key advantage of this staged approach is that with appropriate compiler optimization, no run-time execution cost is introduced, encouraging the use of this library to provide dimension checking in performance-critical code. Nonetheless, the core feature set is not too distinct from other libraries.
- **Python**: Astropy (`www.astropy.org`) and Pint (`github.com/hgrecco/pint`) are the two most popular libraries for Python. They both implement the Quantity pattern in a very similar fashion but their features set and syntax differ greatly.
- **C#**: UnitsNet (`github.com/angularsen/UnitsNet`) is a very popular library. It provides the `IQuantity` interface but is implemented differently to others. The type of the quantity and the value for the seven base dimensions are the only attributes that are defined in the interface. The magnitude (or value) of the quantity is added later as an operation for a specific unit that is calculated. In UnitsNet each specific unit is defined as its own class and utilizes operator overloading to convert between units. The library also employs automatic code generation to produce all the different conversions between different units. Another capable library written in C# is Gu.Units

(`github.com/GuOrg/Gu.Units`). Similar to UnitsNet, it implements the Quantity pattern through the `IQuantity` interface but otherwise provides a more standard implementation of the pattern through a value (`SiValue`) and a unit (`SiUnit`).

The Quantity pattern has been shown to be applicable to mathematical calculations, medical observations and financial conversions. A more detailed and specialized version of this pattern is provided by the Physical Quantity pattern[49] that looks in more depth at the requirements of the physical and mathematical sciences, presenting interfaces to enforce the use of explicit quantity types. The key aspect is that the *research challenges* are not technical in nature, perhaps more work is required to create robust interfaces from the Quantity pattern that engage the respective communities and gather traction. Alternatively even if these well engineered interfaces existed, there might be other reasons that hamper uptake. In order to uncover which, we decided to perform a survey on users.

4 | UNDERSTANDING LACK OF ADOPTION

In this section, we concern ourselves with this lack of commonality[33] and perceived lack of popularity[38] by exploring the many *factors* behind the phenomena and *why*, despite having technical solutions that manage units in code bases, they are not being used extensively.[50] Since, to our knowledge this is a first of its kind study in this particular field, an *exploratory-style* of research was adopted. This entailed conducting interviews with experienced developers and scientists who have used UoM libraries or bespoke solutions to handling units. We began with the initial aim of extracting themes from the conducted interviews and validated those themes with the results of a survey involving practitioners from LinkedIn, Reddit, and GitHub.

The outcome of this study is a list of design principles, and a brief comparison with other methodologies that could aid UoM library developers, and the wider community to adopt UoM solutions. Therefore, the study was divided into three phases:

1. Developers, practitioners and scientists were interviewed based on their varying experiences with UoMs. Two of the developers are involved in the JSR Java unit library project. The interview lasted half an hour. A third developer, a data analyst and functional programming expert, was interviewed for 24 min and contacted after replying to a Quora question on UoM libraries. Finally the fourth subject is an experienced researcher in the Department of Physics and Astronomy at Uppsala University. The interview lasted 23 min.
2. This was complemented with a survey of 91 practitioners from the internet who have had experience with managing units (whether with the help of a UoM library or using their own methodology) to determine what characteristics of current UoM libraries assist or hinder scientists from adopting them.
3. A list of commonly occurring characteristics or issues with UoM libraries are identified from both the interviews and surveys. This entails not only problems inherent to UoM libraries (*internal*factors), but other issues not specific to the libraries themselves (*external* factors).
4. Based on the user feedback of both internal and external factors perceived to be stumbling blocks for adoption, recommendations, and suggestions are given to UoM library creators.

In Section 4.1, we describe how the study was conducted, and subsequently, in Section 4.2, we list some of the key factors extrapolated from both the interviews and discussions. We present a list of design suggestions for UoM developers derived from our study, in Section 4.3, but with caveats on their applicability and use.

4.1 | Research methodology

Data gathering was undertaken in three parts. First through semistructured interviews from hand-selected candidates based on their occupations and experiences with UoM libraries. Second through a survey where practitioners within the field could offer insights to validate the detailed responses from the interview candidates. A third part was to encourage debate and discussion on online forums and bulletin boards. The data gathering and qualitative analysis was conducted by the third author and verified by the first.

4.1.1 | Semistructured interviews

1. *Participants:* The interview participants had to have fulfilled some criteria to be eligible for an interview within this field. Namely, practitioners and scientists from industry who have experience in handling UoM were contacted. Practitioners could include developers and programmers who, for example, developed applications using UoM, have developed UoM libraries and have hosted such projects on GitHub, are active within the units community and evangelize the use of UoM, or have developed applications using their own methodology. Moreover, any scientist was eligible for an interview as long as they were aware of concepts such as dimensional correctness and have worked with units programmatically.

   Email contact was used to identify suitable candidates for the interview. In total, four participants were willing to undergo an interview, where two of the four of the participants are specification leads for a popular Java UoM API (interview subjects 1 and 2). The third interview subject is a lead data scientist at one of the largest retailers in the United States and has experience with units and dimensions in the Haskell programming language, and finally an interview was conducted locally with a materials scientist (the fourth interview subject) at Uppsala University.
2. *Method:* The interviews were conducted either face-to-face or via Skype depending on convenience and flexibility. This resulted in a face-to-face interview with the local materials scientist whilst the other three interviews with the developers were conducted via Skype. Consent from the interview subjects to record their voices was taken so that the interview transcripts could be analyzed. As for the interview questions, there were a set of prepared and open-ended questions but there was generally an incomplete script so improvisation could occur and subjects questioned differently as circumstances dictated. The interview participants were also encouraged to talk about the topic in full detail so that as many viewpoints as possible could be captured. The duration for each interview lasted around 23–30 min, sufficient to explore topics and themes without generating fatigue on the interviewees behalf.
3. *Analysis:* Interview transcripts were manually created from the recordings. The transcripts were then analyzed and coded using qualitative data analysis software. This entailed dividing the qualitative data into smaller units, assigning codes to those smaller units of data based on its content, and then assigning those codes to overarching categories. This process began by highlighting text based on a simple UoM ontology and, through an inductive approach,[51] themes emerged based on similarity, difference, frequency, sequence, correspondence, and causation of described activities. According to Saldana,[52] these are outcomes of codings, categorizations and analytic reflections. By categorizing groups of codes into "code categories," related categories naturally coalesce to form themes. In the context of this article, a theme essentially represents a determinant blocking adoption of UoM libraries. These lack of adoption factor themes are then compared with the quantitative data gathered from the survey so that they could be validated. Themes also emerge from the qualitative natured questions in the surveys and from the discussions held on forum-boards. The importance of a theme only become significant when it occurred both in the survey and in the discussions.

4.1.2 | UoM survey

1. *Respondents:* For the UoM survey, the aim was to attract as many practitioners as possible in the data gathering time frame of the study. This was achieved through the following:

   - Practitioners within our immediate professional networks (e.g., university network) were contacted and given the survey. These practitioners typically work at medium-to-large scale organizations and are accessed via LinkedIn, Facebook or email. The practitioners were also encouraged to distribute the survey to their immediate colleagues or to anyone they knew that works or has worked with UoM.
   - Second, the survey was spread out to social media networks such as Reddit and GitHub. This was achieved by posting the survey to relevant "subreddits," subforums and communities where it was possible to find people of varying experiences who use or have used UoM. It was also possible to pinpoint potential survey-takers by finding practitioners who have contributed to UoM projects or have used UoM in one of their projects. Accounting by experience, it was possible to obtain a diverse set of results.

2. *Format:* The aim was to capture all viewpoints, which meant surveying both practitioners that either rely on UoM libraries (adopters) or practitioners that have their own solutions to managing units (nonadopters). This involved

tailoring the survey differently for adopters and nonadopters. The survey was also tailored depending on the respondents position: developer/practitioner, technical team leader (TTL), or manager. This enabled specific data to be captured for each of the distinct roles that they represented. For adopters, questions included (but were not limited to): (1) why nonadopters would not adopt a UoM library, (2) the noted benefits of adoption, (3) and why the chosen UoM solution was not adopted *earlier*. For nonadopters with their own solutions, questions included (but were not limited to): (1) the reasons behind the choice to not adopt one, (2) guidelines in place to prevent unit conversion failures and errors, (3) what type of solution they would like to see, (4) and the degree of testing difficulty with regards to UoM. Additional questions pertaining to respondent roles were present as well. For instance, quantification of UoM errors for managers, the interference of UoM errors and conversions with developers practices for TTL's. The flow of the survey can be seen in the two tables below, Tables 2 and 3:

3. *Data Analysis:* Results of similarly worded questions, for example, **M-N1, TTL-N1, D-N1** were collated together and then presented in single charts. This extends to questions such as **M-Y1,TTL-Y1,D-Y1**, and so forth. As with the interviews, the results from the open-ended questions were thematically analyzed whilst statistical analysis was used to analyse the results from the more quantitative questions. For example, presenting the ordinal scale costs of unit conversion and unit failures, along with the percentage-wise makeup of results.

In total, of the 91 responses a large majority (67%) came from active practitioners (developers/programmers/scientists) who use UoMs, around 25.3% of respondents were TTL's and a few business unit managers (BU's) at 7.7%.

Regarding adopters, a set of predefined options were available for expressing the reasons for not using UoM libraries. Figure 6 shows the results obtained for the 45 adopter respondents (i.e., managers, TTLs, and developers) who could choose more than one option. The results for nonadopter respondents was broadly similar.

4.1.3 | Forum and bulletin board discussions

Another method of gathering considerable amounts of data was to encourage debate and discussion on online forums and bulletin boards such as Reddit and GitHub. This approach was chosen because it is easier to reach out to practitioners and take note of the rationale behind their choices and decisions. It was an alternative for those who were not keen on being surveyed. Some respondents feel constricted by the format of a fixed survey and would rather talk about the topic

TABLE 2 Survey flow for adopters

Managers	
Y1	Why has it not been adopted *earlier*?
Y2	Why do you think others would not adopt them?
Y3	Noted benefits of adoption?
Y4	Provide a quantitative figure on costs saved.
TTLs	
Y1	Why has it not been adopted *earlier*?
Y2	Why do you think others would not adopt them?
Y3	Noted benefits of adoption?
Developers	
Y1	Why has it not been adopted *earlier*?
Y2	Why do you think developers would not adopt them?
Y3	Noted benefits of adoption?
Y4	Would you recommend others to adopt UoM libraries?
Y5	How influential are you to get others to adopt these libraries?
Y6	To what degree is testing easier with a UoM library?

Abbreviation: TTL, technical team leader.

TABLE 3 Survey flow for nonadopters

Managers	
N1	Why has it not been adopted?
N2	How costly are the errors? (Ordinal)
N3	How interfering is it with coders practices?
N4	Could you quantify the error in your local currency?
N5	Would you consider adopting a UoM library in hindsight?
TTLs	
N1	Why has it not been adopted?
N2	What guidelines are in place to prevent unit conversion failures?
N3	How interfering are UoM errors?
N4	What type of solution do/would you prefer?
N5	Would you consider adopting a UoM library? (Ordinal)
Developers	
N1	Why has it not been adopted?
N2	How difficult is testing UoM code?
N3	How interfering are UoM errors?
N4	What solution would you like to see?
N5	Would you consider adopting a UoM library in hindsight?
N6	How frequent are UoM errors?

Abbreviation: TTL, technical team leader.

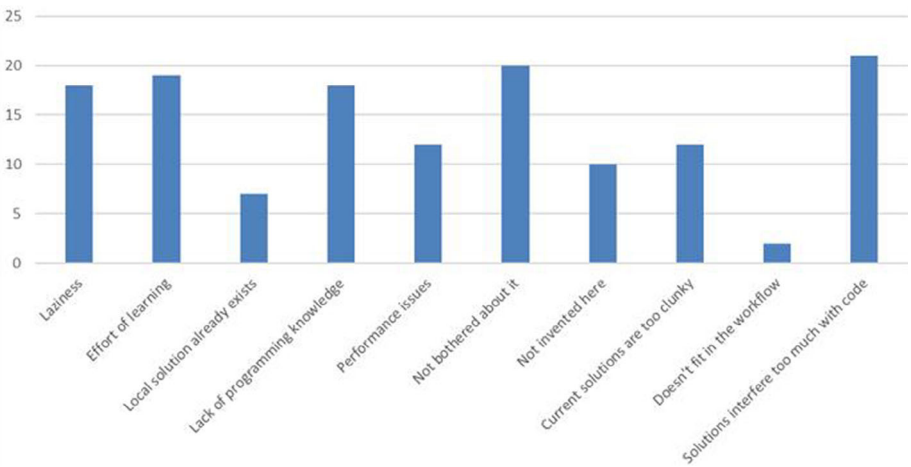


FIGURE 6 The reasons adopters gave for not using UoM libraries [Colour figure can be viewed at wileyonlinelibrary.com]

in a less constrained manner, thus gaining more potential data that otherwise would not have been captured. In total, the forum discussions yielded 67 qualitative data points which were collated into a single document. Similar to the analysis in Section 4.1.1, the discussions were *thematically analyzed using qualitative coding* and the themes were validated with the data from the survey and interviews.

4.2 | Lack of adoption factors

In this section, the main characteristics and factors behind the lack of adoption of UoM libraries are listed. As mentioned earlier, every factor is a theme, stemming thematically from the interviews and are reoccurring phenomena/themes that

appear in both the interviews and discussions. The occurrence of a theme *in more than one part* of the data gathering process (survey, interviews, forums) qualifies the theme to be a noteworthy factor to talk about. We have grouped the factors into three sections. The first describes factors that impede the use of UoM libraries, the second looks at shortcomings in their implementation, while the third factor relates to the wider context of their use.

4.2.1 | Ignorance and apathy

There is a considerable lack of awareness surrounding UoM libraries. Library developers are unaware of each other's efforts. Practitioners are aware of unit failures and their implications, but what they generally do not know is that there exists libraries that can handle and prevent these failures to some degree.

Unawareness A main and reoccurring phenomena throughout the data gathering and analysis process was unawareness of any UoM solution, which could be a major reason behind the lack of social push for the adoption of UoM libraries. It seems many have not even considered the possibility of UoM libraries existing. Some of the survey nonadopter respondents had not even heard or thought of it before the survey was launched. The following are respondent and interview comments that support this claim:

- *"Ignorance of UoM solutions; that is, where and when they should be used. This is not [a] lack of programming knowledge, but rather a lack of knowledge that a problem and solutions even exists."*
- *"I didn't even know there were libraries that did this for you"*
- *"Because their existence was not yet known to me and my coworkers at the time."* (before adoption)
- *"Not aware."*, *"Didn't think of it."* and *"Didn't think of the possibility."*
- *"Units of measurement are fully integrated into the F# language. It would be interesting to see what uptake is like in that community versus others. In other languages, where you will never see it unless you look for it."*
- *"I recently helped develop a major code for the simulation of fluid flow in Fortran and nobody ever asked for units anywhere, but everyone was really interested in performance."*
- *"I think awareness is a major issue. I hadn't heard about it up until about 2 weeks ago."*

To further aid these remarks, interview subject 3 stated in a straightforward manner that most people are simply unaware of the existence of UoM libraries. Even if practitioners work with units in their code, it is very unlikely that they have heard of 'UoM libraries' specifically. This was evident in the discussion forums where it was a pervasive theme throughout the exchanges. Moreover, it seems as if some of the nonadopter practitioners only had a rudimentary understanding of UoMs (e.g., knowledge limited to only converting units without regards to dimensional correctness) and were not concerned or were unaware that UoM solutions existed which could handle these issues to some extent. We end this subsection with a response that could be a determining reason behind the prevalent lack of awareness: *"Not part of std libraries for the most part"*.

Lack of concern/indifference The general lack of concern is also an apparent phenomenon, a sizeable amount of nonadopter respondents, which totaled to around 33%, were simply not concerned. In addition, 44% of adopter respondents believed that insufficient concern was a factor behind the lack of adoption of UoM libraries.

- *"Perception that they are not needed."*
- *"This isn't an important problem for me."*
- *"I am not sure why it wasn't considered. I think it isn't much of a problem."*
- *"I've never felt a need for them."*

No gain/need in use case It seems as if a large majority of nonadopters simply believed that they had nothing to gain from including a dependency to a UoM library. According to some, these libraries could cause even more harm than they profess to solve. For one respondent, it was not even seen as a crucial component, but more of a "nice-to-have" feature. This depends largely on the use case but barring applications that are centered around physical and mathematical calculations, it seems (according to interview subject 2 and some of the respondents) that UoM libraries have too narrow of a use case to truly reap the rewards from their inclusion.

- *"People either don't really feel the immediate need to have something like this or they don't even realize that there are unit libraries out there."*
- *"Narrow requirements."*
- *"Java projects do too little numerical calculations and with only a few units, so nobody bothers to do it strictly with units."*
- *"Not really applicable to our problem domain."*
- *"Manual conversion functions covered most common cases."*

From these claims, respondents agree that UoM libraries are best suited for complicated science or complicated dimensions and analyses and not for a lot of mundane data science or software development.

Impetus/lack of familiarity A reason to include DY5 (how influential are you to get others to adopt these libraries) was to gauge the impact that a social network had on encouraging the adoption of a technology. From the adopter respondents, 42.5% somewhat influential and 16.1% of respondents are slightly influential. 25.8% of respondents were not at all influential.

Although the idea of a UoM library might seem intuitive and straightforward, the technicalities and inner workings of such a library would in theory be hard to understand and apply effectively. UoM libraries are not widely known and adopted yet and the lack of existing traction (impetus) could be slowing adoption. The reasons for this could be (according to interview subject 2) that people who are both influential and well-connected have not pushed for, or evangelized, unit libraries as much as others have for alternative technologies (such as the Nix package manager in the Haskell community, in the interview subject's example).

- *"For almost everything else you're going to have to go out of your way to get some library or some tool that nobody knows yet and have to start using it. So there's a big barrier to entry."*
- *"Where you have one or two people who are connected in the right way and excited about something and want to show it off and then they can take off within a company or community. And so you end up with technologies going together where there's nothing technical or very little technical, but the people who like them are the same."*
- (Respondent comment) *"Inertia."*
- *"I have pushed for UoM types at two companies with limited success (autonomous vehicles), it is hard to get developers to use UoM in their internal application logic."*
- *"Most developers are not familiar with it (scared by compiler errors)."*

Tradition According to the fourth interview subject, sticking to what already works is guaranteed "safety." Tradition or a fear of change, especially rooted in scientific communities could lead to a lack of adoption of UoM libraries due to existing solutions already satisfying requirements, therefore there is little incentive for change.

- *"I would say the most interested users would be scientists, but scientists consider code something secondary in their work; sometimes even a burden. Most scientific code is not touched anymore after the article is published which does not contribute to the situation."*
- *"And the reason UoM libraries were not being used? I don't know maybe just tradition, no one is being paid to develop code more or less but you're being paid to get research grants to make progress in a specific field and you make code to solve the problem."*

Not-invented-here syndrome NIH syndrome with regards to UoM is the tendency of developers to reject external solutions in favor of already existing unit solutions based on suitability, convenience and a myriad of other factors. Below are a few anecdotes from the discussions and surveys:

- *"We made our own library."*
- *"Custom solution."*
- *"The only thing I use is std::chrono. And just because it is easily available."*

- *"Why build your own instead of using an existing one? Same reason people roll their own StringUtil libraries—Not Invented Here syndrome (or company policies that restrict external libraries)."*
- *"I like the java.time library, and would like something similar for other units."*

4.2.2 | Technical internal factors

The core issue that these libraries aim to solve is something that is relatively easy to understand and familiar. However, as Bekolay argued *"making a physical quantity library is easy but making a good one is hard"*.[33] The problems arise when trying to make a more complete library, including more units, more operators, effective conversions, good error messages, efficient and accurate. It is far more challenging than a robust implementation of the Quantity pattern.[11,39]

Effort of learning, ROI, and overhead Even if you have a project for which UoM checking would be suitable, the perceived effort required to master these libraries, which may not provide a ROI (return on investment), is a deterrent for adoption. This concern was evident from the survey where a considerable amount of adopters (42%) and nonadopters (32%) listed "effort of learning" as a reason behind the lack of adoption. A few excerpts from the discussions and surveys can be seen below:

- *"Investment of effort required to use such a library."*
- *"Adding a UoM framework would add extra intellectual encumbrance that in the end would likely cause more bugs than it would help prevent."*
- *"Lack of knowledge that this is a problem and/or value of using them not perceived better than pain of learning a new library."*
- *"Reluctance to depend on external libraries for a low-priority feature. We started using them when we switched to F#; being built in to the language was key."*
- *"For the few units that are built-in, there is virtually no gain to having the unit checked. This is because the value passes through numerous generic components that don't care about units: database interfaces, message passing interfaces, user interface components, IO interfaces, and so forth."*
- *"No-effort "sweet spot" solution not easily found."*
- *"I tried working with Boost.Units multiple times. But usually I failed. And the reason in all these cases was that simply setting up the whole system took more time than I had to invest."*
- *"They cause more pain than they relieve."*
- *"Most people don't bother wrapping a long time stamp in a java.time.Instant, or a user id String in a UserId object, or a UUID String in a UUID object, so going even further and using a library for wrapping other units is just too much effort."*

Run-time unit handling This factor is particularly interesting as some respondents are of the belief that UoMs are a purely compile-time construct, quoting a respondent that *"dimensions cannot be a run-time concept as they are intimately related [to] the types of variables used"* and that *"units actually don't need to exist at run-time at all. They only need to exist at compile-time to check them and generate the correct conversions with their multiplier,"* but from the survey and discussions, it seems that handling units at run-time was a focus and specific requirement to use UoM libraries. The first and second interviewees state that run-time unit handling is harder to accomplish, especially in Java with the manner by which generics are compiled. In practice the use of generic types ensures that run-time type information is not available. Respondent anecdotes below state what is lacking in UoM solutions and why having run-time handling of units could be useful:

- *"No support for units that are modifiable at run-time as opposed to compile-time."*
- *"Where I work, Boost.Units isn't especially useful because of the design that the units of interest are known and fixed at compile-time."*
- *"In our product line, our users may very well have one file (or one attribute of a multiattribute file) whose units are "kg/m^3," another whose units are "g/cc" and a third whose units are "degrees Celsius." We therefore need to be able to operate on units (parse, convert, multiply or divide, and check for dimensional consistency) at run-time, not compile-time."*

Performance How UoM libraries perform is a large concern for a great amount of nonadopters. As shown in Wand and O'Keefe,[26] unit checking can be performed at compile-time without a run-time cost. However, with libraries, their implementation requires boxed values rather than the standard primitive entities. When units are not part of the language and UoM libraries are employed then there is a cost at both compile-time and run-time.

An unnecessary run-time cost would be when every temporary variable in a matrix multiplication has to have a unit annotation. This drastically increases computational complexity when UoM checking could have ensured correctness at compile-time. For applications that carry out lots of calculations, their operation will be affected more as boxed values with types would have unnecessary performance overheads. The cost of having unit safe code comes at the expense of performance, this concern is especially apparent for numerical simulations, where more priority is placed on making sure simulation code works first rather than being safe from unit errors. The following remarks below emphasize this as a factor behind the lack of adoption:

- *"Template instantiation baggage is too costly for benefit."*
- *"None of these issues were addressed by adoption and were all a tax on subsequent use."*
- *"Robustness against mistakes, compile-time checking"*
- *"Compilation overhead not worth the benefit of just unit-testing math code"*
- *"Even if you had a linear algebra library that supported UoM, I fear that compile-times could grow nonlinearly with the number of dimensions in your problem, since the compiler would have to derive a UoM type for every temporary variable."*

Loss of precision Converting between units introduces errors and reduces precision for very large and/or very small models.[53] Even the most efficient unit converters[40] do not attempt to minimize round-off errors. For some UoM libraries, the underlying variable type might be set thereby limiting a programmers choice.

- *"I dislike that Boost.Units defaults to double for underlying storage, and the library does not let you pick the units or scale of the backing store (e.g., lengths are always stored in meters). This leads to loss of precision when you try to use something other than doubles."*
- *"Boost.Units is ridiculously over complicated and many of its conversions are inaccurate and lousy."*

Dimensional inconsistencies For an equation to be dimensionally consistent, the units on the LHS and RHS must equate. It goes without saying that one of the most important reasons to use a UoM library is to ensure that programmed equations have the same dimensions. Conversion factors can be applied to either side of the equation to ensure unit conformance. If the units in an equation do not resolve dimensionally then the equation is inconsistent, which could possibly lead to catastrophic implications depending on the use case.

- *"I've had some very confusing dimensional analysis moments when the units didn't negate correctly, no matter how many different aliases or permutations of equivalent units I included. Something like problems dealing with flow can give really ugly units."*
- *"For instance, a unit of N will not cancel when divided by a unit kg unless you use kg * m/s^2 in lieu of N. This isn't so bad for simple problems, but when you have flow equations or EM equations, it gets messy fast without the nice composite unit aliases."*

4.2.3 | External factors

Casting aside factors relating to ignorance and weak technical implementations, there are certain factors that will impede library adoption regardless. Modern systems are not built in a vacuum; they rely on legacy systems, databases and many other components that are unlikely to support UoM annotations without costly updates.

Integration The cost of porting existing, "working code" to communicate with UoM libraries is a major challenge. Both the overhead and the cost of dependency of a UoM library is a major factor for many respondents. In high performance computing applications, composed of mostly physical and engineering calculations, the main priority goes towards the best numerical algorithms and implementations. Currently these exist outside of UoM library support in most languages

surveyed. One respondent commented that Boost.units[20] `quantity<T>` does not preserve standard-layout, triviality and POD-ness (plain old data) of `T`, which are critical requirements for their domain. These type restrictions, for example, might make performing algebraic operations a challenging task.

- *"Not supported/compatible with the linear algebra library that we use (Eigen3)."*
- *"We depend on a number of libraries (PETSc and firedrake, in the past deal.II, Eigen, trilinos, and so forth) that interact very poorly with UoM libraries. We would use them if we could."*
- *"Too much friction using with non-UoM libraries."*
- *"Obvious solutions exist on both ends of the spectrum (convenient: just use floats; safe: define a custom type per unit that only implements the desirable conversions), and fitting something in between in a way that interacts well with the type system is hard in our language of choice."*
- *"Perhaps units are too little of a concern for a, let's say, "more general" codebase where many things are without units."*

When collating the Y1 and N1 questions for each role in the survey, around 46% of adopter respondents and 38% of nonadopter respondents answered that UoM solutions interfere too much with currently existing code.

These anecdotes highlight that the unit types provided by UoM libraries are not standardized yet, hence the friction with non-Unit libraries. It is therefore easy to understand why some nonadopters would rather write unit-less code since using common variable types would be, by default, more accepting of other interfaces.

Nonintuitive A few respondents felt that UoM libraries were not intuitive to use *in some programming languages*. From a software engineering perspective, if a library or tool is difficult to use then it is unlikely to be selected. Respondent anecdotes emphasize this point:

- *"Very few languages have the affordances to make numbers with units both statically type-checked and as ergonomic to pass in and out of functions as bare numbers."*
- *"It feels very tedious to write units every[time] in the program."*
- *"Make it as idiomatic to use an UoM library as possible: If the language allows, these are prime examples where operator overloading makes sense."*

It would seem as if respondents agree that in languages that have numerical typing systems similar to the hierarchical type classes of Haskell that support operator overloading, using a UoM library would be very natural. However, in languages where binary math operators would no longer be used and types would have to be explicitly written (e.g., Java), extra code friction would result in less use.

Type restrictions Linked closely to integration and performance problems, respondents continually claim that one of the drawback of UoM libraries is that they are affected by type restrictions. Interview subjects 1 and 2, who were lead API developers for a Java UoM API acknowledged the boilerplate nature of UoM code, pinning the problem on the verbose nature of Java as a weakness. Both Kotlin and Scala adopt extensions of the UoM library under the hood, Physikal[54] and Squants,[55] respectively. Thereby incorporating a few additional features that Java so far has not adopted, and hiding a lot of boilerplate code under the wrapper. A few anecdotes are listed below which give light to this drawback.

- *"All UoM libraries fail miserably at supporting anything but simple computations. I know of no linear algebra library (e.g., Eigen) that allows UoM types to be preserved across computations."*
- *"You can reinterpret cast your way to success (provided that the UoM library merely wraps arithmetic types) to adapt to a linear algebra library's API, but that boilerplate code is ugly, tedious, and it defeats the whole purpose of UoM types."*
- *"I believe that many UoM libraries are nonidiomatic in their use (either by library-design, or because the language does not allow custom datatypes to be used in the same "normal" way as "built-in" numeric types). This extends to, for example, not being able to use a UoM library with other libraries (that e.g., require plain ints or doubles as input), making the adoption of UoM libraries a chicken-and-egg problem."*
- *"The majority of units are user-defined, and thus cannot be checked statically at compile-time."*

Difficulty of use This factor may differ vastly among the many UoM libraries. While the idea of using a UoM library might seem appealing and straightforward, a few anecdotes from respondents reveal that this may not be the case. At first glance, a UoM library might seem easy to use and include in a software project, but the inner workings of the UoM library could increase the complexity of a project. It is of particular interest to note from some respondents would rather spend more time writing unit tests that would catch UoM errors, rather than relying on a unit library to detect errors at compile-time. It is also likely that respondents do not want to be burdened with annotating UoM throughout their code. It is clear that there is a trade-off when using a UoM library.

- *"The solutions you can create without language support are always lacking and annoying to use."*
- *"Boost.Units has an insane amount of templated magic going on. And if you need to make proper use of the library, you need to dig deeply into it."*
- *"Using units should make things simpler and more understandable, not harder."*
- *"I could use the same time to write tests and that would really find and prevent errors and at the same time not introduce a crazy complicated library every other developer in my team would have to deal with."*

4.3 | Design suggestions for unit libraries

In this study, we have attempted to evaluate the design and usage of UoM libraries based on a design science approach.[56] There are certain suggested design principles that arise from the themes presented in the previous section.

- Developers must favor simplicity in their design of unit libraries to reduce verbosity and boilerplate code. It is worth considering dispensing with clunky, generic style OOP programming by wrapping these approaches entirely in syntactic sugar or through the support of an IDE. One respondent mentioned that a good standard for a UoM library to be "easy" to use would be if it were no more complicated than `Java.Time` or `std::chrono`. Unfortunately few of the UoM libraries meet this standard.
- It is vital to design a UoM library to be as idiomatic as possible. A UoM library can be restrictive if it does not support nominative inputs. This means allowing language features such as operator overloading if the language permits it (such as in Haskell and Rust).
- Unit code must return meaningful error messages in case of any unit inconsistencies.
- UoM libraries should also provide some degree of support for custom unit types and user defined dimensions and units.
- UoM libraries should allow support for mixing of UoM types with different underlying storages, just as a double and float can often be multiplied. Multiplication of two UoM types with different backing stores should be allowed and the result should follow the type promotion rules of the underlying programming language. Care must be taken to combat precision loss.
- UoM libraries should be easy to adapt to other libraries. An adapter layer or component for common libraries such as Eigen[57] would therefore be a good solution and middle ground for UoM libraries and other APIs which do not readily accept UoM types.

The design suggestions can be summarized as follows: UoM types need to be as straightforward to use as arithmetic types or at least as close as possible. Alas this cannot be easily achieved with libraries as language extensions or IDE support is often required to facilitate the additional syntax. C++ is the solitary language for which a de facto standard library has emerged, and even then it has implementation limitations as revealed by our study.

A subtle observation is that some practitioners thought UoM should be used to check for correctness at compile-time, as demonstrated in Figure 1, while others required units to be present at run-time so that their programs could correctly manipulate inputs with varying base units. Physical entities may have fixed conversions but money conversions, for instance, could require the usage of a web service to perform up to date conversions on currencies only known at run-time. This has profound implications on how the library is implemented, and the potential run-time cost. Compile-time checking can be achieved through static overloading, or Java generic instantiations. While run-time checking is achieved through overriding. We demonstrate in Figure 7 both techniques using the Java class structure for the UoM Helen, a humorous measurement of beauty based on Helen of Troy who had a "face that launched a thousand ships," along with

Overloading

```
public interface I_Beauty
{
    float get_value ();
    void set_value (float h);
    void add (Helen h);
    void add (MiliHelen mh);
}

public class Helen implements I_Beauty
{
    private float h;
    public Helen(float h) { this.h = h; }
    public float get_value() { return h; }
    public void set_value (float h) { this.h = h; }
    public void add (Helen h) { this.h = this.h + h.get_value(); }
    public void add (MiliHelen mh) { this.h = this.h + (mh.get_value()/1000); }
}

public class MiliHelen implements I_Beauty
{
    private float mh;
    public MiliHelen(float mh) { this.mh = mh; }
    public float get_value() { return mh; }
    public void set_value (float mh) { this.mh = mh; }
    public void add (Helen h) { this.mh = this.mh + (h.get_value()*1000); }
    public void add (MiliHelen mh) { this.mh = this.mh + mh.get_value(); }
}
```

Overriding

```
public interface I_Beauty
{
    float get_value ();
    void set_value (float h);
    void add (I_Beauty b);
}

public class Helen implements I_Beauty
{
    private float h;
    public Helen(float h) { this.h = h; }
    public float get_value() { return h; }
    public void set_value (float h) { this.h = h; }
    public void add (I_Beauty b) {
        if (b instanceof Helen)
           h = h + b.get_value();
        else // b instanceof MiliHelen
           h = h + (b.get_value()/1000); }
}

public class MiliHelen implements I_Beauty
{
    private float mh;
    public MiliHelen(float mh) { this.mh = mh; }
    public float get_value() { return mh; }
    public void set_value (float mh) { this.mh = mh; }
    public void add (I_Beauty b) {
        if (b instanceof MiliHelen)
            mh = mh + b.get_value();
        else // b instanceof Helen
            mh = mh + (b.get_value()*1000); }
}
```

FIGURE 7 The UoM Helen implemented both statically and dynamically

the addition operator. We have to explicitly perform the object test at run-time in order to achieve the same behavior as the version using overloading. Even for a single UoM with a prefix, namely, `mili`, we have distinct object level behavior depending on whether we want compile-time or run-time checking. The signature differs for the method `add`depending on our use case. Combining both semantics, even with the compact Quantity Pattern representation, would double the amount of syntax required and further complicate usage. Dynamically typed languages, such as Ruby or Python, will by definition perform checking at run-time.

5 | ALTERNATIVE METHODS FOR UNIT CHECKING

When UoM exist as a construct there is no run-time overhead as the checking can be performed at compile-time. They can provide a simpler syntax and avoid hard to detect run-time errors. Moreover, the compiler could optimize expression evaluation to limit unit conversions and rounding errors. One of the respondents specifically commented on their reluctance to use libraries but having adopted units once they switched over to F#, see Section 4.2.2 (Effort of Learning, ROI and Overhead). Such strong static checking might be sought by some, but could be a hindrance to others who require units to be defined at run-time. Unit mismatch and conversion errors can be detected by UoM libraries but avoiding programming style errors requires further discipline that a conventional static checker provides. Such errors are caused by violations of standard type systems, for example, when an intermediate variable is used with different units. These were found to account for 75% of inconsistencies in the study of 5.9M lines of code.[43]

It was clear that even the best libraries currently cause significant performance issues while not being relevant for most developers. Lightweight component-based approaches that do not burden the scientific programmer with the need to annotate each statement have been proposed. Damevski[58] postulates that unit libraries are too constraining and incur an annotation or migration burden. He argues that units of measurement should be inserted in software component interfaces. There is some anecdotal evidence in the many quotes of Section 4.2 to support this approach. His algorithm attempts to resolve UoM types at run-time so that if the types of the method's parameter and argument are compatible to each other than type conversions occur. Another light weight methodology was presented by Ore et al.[42] that uses an initial pass to build a mapping from attributes in C++ shared libraries to units. The shared libraries contain unit specifications so this mapping is used to propagate into a source program and detect inconsistencies. Their compile-time algorithm leverages dimensional analysis rules for arithmetic operators to great effect.[59]

A component interface based discipline means that the consequences of local unit mistakes are underestimated. On the other hand, it allows diverse teams to collaborate even if their domain specific environments or choice of unit systems were, to some extent, incompatible. More importantly it would have been sufficient to have caught the Mars Climate Orbiter error.

A final means of ensuring UoM checking is through automated unit testing. It is fairly common nowadays to develop tests alongside code, not only for the purpose of test driven development but also to ensure maintainability through refactoring. Including UoM tests requires no extra tool support and will not affect the eco-system. Spending time writing unit tests would equate to adding unit annotations without the introduction of a library, see Section 4.2.3 (Difficulty of Use). However, the UoM knowledge will be localized to each particular unit so the slight implementation cost comes at the expense of potentially incomplete checking. Lightweight methods often provide "good enough" detection without the drawbacks.

6 | CONCLUSIONS

The software engineering benefits of adopting unit checking and automatic conversion support is indisputable. As every line of code can be subjected to a large range of values and dependencies, unit errors are bound to arise. Providing unit support is an obvious way to mitigate for such human error. UoM systems help manage complexity, allowing the programmer to rely on the checker to ensure correctness and not on themselves or their colleagues. Moreover, unit annotations are relatively stable to program reorganization, refactoring will rarely require annotation changes unless the underlying data structures are also modified. This stability ensures the maintainability and scalability of UoM annotations within potentially safety-critical code.

The initial study presented in this article summarized the large but little known area of UoM libraries, highlighting an abundance of similar contributions but without a critical mass being reached to achieve dominant libraries for most programming languages. Practitioners are aware of UoM failures and their implications, but what they generally do not know is that there exists libraries that can handle and prevent these failures to some degree. However, UoM are hard to define, cumbersome syntactically, costing developer time to annotate, and expensive dynamically. We confined our study to mainstream languages but this study should be expanded to include popular spreadsheets,[28] mathematical tools and domain specific scientific notations,[40,60] in which there is plenty of scope to successfully leverage compile-time unit checking and optimal dimension conversion.

In order to understand the proliferation of libraries, we choose to interview developers, practitioners and scientists in an exploratory style to uncover their views on the topic. The interviews, surveys and online discussion boards have confirmed some of our original hypotheses,[38] namely that these libraries do not satisfy the core requirements of the scientific programming community, allowing programmers to manage UoM in an indistinguishable language agnostic fashion. Interview subjects felt that UoM libraries were inconvenient: they did not interact well with the eco-system, had performance issues, required effort to learn and costly rewrites to support. This explains the many reasons given for lack of adoption shown in our study. UoM libraries are best suited to run-time checking where performance is not a key condition but robustness is.

ACKNOWLEDGMENTS

The University of Western Australia Department of Computer Science and Software Engineering for providing a warm escape, as part of a Matariki exchange fellowship, while this article was drafted. Dr Duggan for suggesting the Helen UoM, and the reviewers for their insightful remarks and constructive suggestions.

ORCID

Steve McKeever https://orcid.org/0000-0002-1970-2884

REFERENCES

1. Stephenson A, LaPiana L, Mulville D, et al. Mars climate orbiter mishap investigation board phase 1 report; 1999. https://llis.nasa.gov. Accessed October 14, 2020.
2. Dieterichs H. Units of measure validator for C#; 2012. https://www.codeproject.com/Articles/413750/Units-of-Measure-Validator-for-Csharp. Accessed April 15, 2020.

3. Neumann PG. Illustrative risks to the public in the use of computer systems and related technology. *SIGSOFT Softw Eng Notes*. 1992;17(1):23-32. https://doi.org/10.1145/134292.134293.
4. Vallecillo A, Morcillo C, Orue P. Expressing measurement uncertainty in software models. Paper presented at: Proceedings of the 10th International Conference on the Quality of Information and Communications Technology. Lisbon; 2016:15-24.
5. Joint Committee for Guides in Metrology (JCGM). International vocabulary of metrology, basic and general concepts and associated terms (VIM); 2012. https://www.bipm.org/en/about-us/. Accessed April 15, 2020.
6. Bureau International des Poids et Mesures The international system of units (SI). https://www.bipm.org/en/measurement-units. Accessed April 15, 2020.
7. Sonin AA. *The Physical Basis of Dimensional Analysis. Technical Report*. Boston, MA: Massachusetts Institute of Technology; 2001.
8. The National Institute of Standards and Technology International System of Units (SI): base and derived; 2015. https://physics.nist.gov/cuu/Units/units.html. Accessed October 2, 2019.
9. Bureau International des Poids et Mesures SI brochure: the international system of units (SI). 9th, *Dimensions of Quantities*. France: BIPM; 2020. https://www.bipm.org/utils/common/pdf/si-brochure/SI-Brochure-9.pdf. Accessed April 15, 2020.
10. Cmelik RF, Gehani NH. Dimensional analysis with C++. *IEEE Softw*. 1988;5(3):21-27. https://doi.org/10.1109/52.2021.
11. Fowler M. *Analysis Patterns: Reusable Objects Models*. Boston, MA: Addison-Wesley Longman Publishing Co., Inc; 1997.
12. Ore JP, Elbaum S, Detweiler C, Karkazis L. Assessing the type annotation Burden. Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering; 2018:190-201. https://doi.org/10.1145/3238147.3238173.
13. TIOBE The importance of being earnest index; 2020. https://www.tiobe.com/tiobe-index/. Accessed April 15, 2020.
14. Apple. Swift Open Source; 2020. https://swift.org. Accessed April 15, 2020.
15. Microsoft. F# Software Foundation; 2020. https://fsharp.org. Accessed April 15, 2020.
16. Milner R. A theory of type polymorphism in programming. *J Comput Syst Sci*. 1978;17:348-375.
17. Kennedy A. Dimension types. programming languages and systems—ESOP'94. Paper presented at: Proceedings of the 5th European Symposium on Programming. Edinburg; 1994;788:348-362.
18. Java Community Process JSR 363: units of measurement API; 2016. https://jcp.org/en/jsr/detail?id=363. Accessed July 2, 2018.
19. Keil W. Eclipse UOMo project; 2020. http://www.eclipse.org/uomo/. Accessed April 15, 2020.
20. Schabel M, Watanabe S. Boost C++ Libraries, Chapter 43 (Boost.Units 1.1.0); 2020. https://www.boost.org/. Last Accessed April 15, 2020.
21. Nemerle Programming Language; 2018. http://nemerle.org/. Accessed April 15, 2020.
22. Eliasen A. Frink programming language; 2020. https://frinklang.org. Accessed April 15, 2020.
23. Karr M, Loveman DB. Incorporation of units into programming languages. *Commun ACM*. 1978;21(5):385-391. https://doi.org/10.1145/359488.359501.
24. Gehani N. Units of measure as a data attribute. *Comput Lang*. 1977;2(3):93-111. https://doi.org/10.1016/0096-0551(77)90010-8.
25. Dreiheller A, Mohr B, Moerschbacher M. Programming pascal with physical units. *SIGPLAN Notes*. 1986;21(12):114-123. https://doi.org/10.1145/15042.15048.
26. Wand M, O'Keefe P. Automatic dimensional inference. *Computational Logic - Essays in Honor of Alan Robinson* 1991: 479-483. MIT Press.
27. Kennedy A. Types for units-of-measure: theory and practice. *Central European Functional Programming School - Third Summer School*. Berlin, Heidelberg: LNCS; 2009;268-305.
28. Antoniu T, Steckler PA, Krishnamurthi S, Neuwirth E, Felleisen M. Validating the unit correctness of spreadsheet programs. Proceedings of the 26th International Conference on Software Engineering; 2004:439-448.
29. Hilfinger PN. An ada package for dimensional analysis. *ACM Trans Program Lang Syst*. 1988;10(2):189-203. https://doi.org/10.1145/42190.42346.
30. Mayerhofer T, Wimmer M, Vallecillo A. Adding uncertainty and units to quantity types in software models. Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering; 2016:118-131. https://doi.org/10.1145/2997364.2997376.
31. Allen E, Chase D, Luchangco V, Maessen JW, Steele GL. Object-oriented Units of Measurement. Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications; 2004:384-403. https://doi.org/10.1145/1028976.1029008.
32. Gibson JP, Méry D. Explicit modelling of physical measures: from Event-B to Java. Paper presented at: Proceedings of the 1st International Workshop on Handling IMPlicit and EXplicit knowledge in formal system development. Xi'An, China; 2017:64-79. https://doi.org/10.4204/EPTCS.271.5.
33. Bekolay T. A comprehensive look at representing physical quantities in Python. *Sci Comput Python*. 2013. https://pyvideo.org/scipy-2013/a-comprehensive-look-at-representing-physical-qua.html.
34. Zaghi S. Fury library comparison; 2020. https://github.com/szaghi/FURY#libraries. Last Accessed April 15, 2020.
35. Unit conversion shootout; 2012. https://github.com/JustinLove/unit_conversion_shootout. Accessed April 15, 2020.
36. Moene M. . Physical Units; 2013. https://github.com/martinmoene/PhysUnits-CT. Accessed April 15, 2020.
37. Baumann T. Quantities, Libraries in other programming languages. Online https://github.com/timjb/quantities/wiki/Links; 2017. Last Accessed 15th April 2020.
38. Bennich-Björkman O, McKeever S. The next 700 unit of measurement checkers. Paper presented at: Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering. Boston; 2018:121-132. https://doi.org/10.1145/3276604.3276613.
39. McKeever S, Paçaci G, Bennich-Björkman O. Quantity checking through unit of measurement libraries, current status and future directions. Paper presented at: Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development. Prague; 2019. https://doi.org/10.5220/0007524704410447.

40. Cooper J, McKeever S. A model-driven approach to automatic conversion of physical units. *Softwa Pract Exper*. 2008;38(4):337-359. https://doi.org/10.1002/spe.828.
41. Jiang L, Su Z. Osprey: a practical type system for validating dimensional unit correctness of C programs. Paper presented at: Proceedings of the 28th International Conference on Software Engineering. Shanghai; 2006:262-271. https://doi.org/10.1145/1134285.1134323.
42. Ore JP, Detweiler C, Elbaum S. Lightweight detection of physical unit inconsistencies without program annotations. Paper presented at: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. Santa Barbara; 2017:341-351. https://doi.org/10.1145/3092703.3092722.
43. Ore JP, Elbaum S, Detweiler C. Dimensional inconsistencies in code and ROS messages: a study of 5.9M lines of code. Paper presented at: Proceedings of the International Conference on Intelligent Robots and Systems. Vancouver; 2017:712-718.
44. Hong NC. Choosing a repository for your software project; 2018 https://software.ac.uk/resources/guides/choosing-repository-your-software-project. Accessed April 15, 2020.
45. Stackify. Top source code repository hosts: 50 repo hosts for team collaboration, open source, and more; 2017. https://stackify.com/source-code-repository-hosts/. Accessed April 15, 2020.
46. Comparison of source code hosting facilities; 2020. https://en.wikipedia.org/wiki/Comparison_of_source_code_hosting_facilities. Accessed April 15, 2020.
47. A common core package for Astronomy in Python; 2020. https://www.astropy.org. Accessed April 15, 2020.
48. Olson M. *The Logic of Collective Action: Public Goods and the Theory of Groups, Second Printing with New Preface and Appendix. 124*. Cambridge, Massachusetts: Harvard University Press; 2009.
49. Krisper M, Iber J, Rauter T, Kreiner C: Towards a pattern language for quantities and units in physical calculations. Paper presented at: Proceedings of the 22nd European Conference on Pattern Languages of Programs. Irsee, Germany; 2017:9:1-9:20. https://doi.org/10.1145/3147704.3147715.
50. Salah OA, McKeever, S. Lack of adoption of units of measurement libraries: survey and anecdotes. Paper presented at: Proceedings of the International Conference on Software Engineering, *Software Engineering in Practice*. Seoul; New York, NY: ACM; 2020. https://doi.org/10.1145/3377813.3381359.
51. Oates BJ. *Researching Information Systems and Computing*. California: Sage Publications Ltd; 2006.
52. Saldana J. *The Coding Manual for Qualitative Researcherson*. 3rd ed.California: SAGE Publications Ltd; 2015.
53. Wikipedia: floating-point arithmetic - accuracy problems; 2020. https://en.wikipedia.org/wiki/Floating-point_arithmetic#Accuracy_problems. Accessed April 15, 2020.
54. Physikal. A units of measurement extensions library; 2020. https://github.com/unitsofmeasurement/Physikal. Accessed April 15, 2020.
55. Squants. The Scala API for quantities, units of measure and dimensional analysis; 2020. https://www.squants.com. Accessed April 15, 2020.
56. Hevner AR, March ST, Park J, Ram S. Design science in information systems research. *MIS Q*. 2004;28(1):75-105.
57. Jacob B, Guennebaud G. Eigen: a C++ template library for linear algebra; 2018. http://eigen.tuxfamily.org. Accessed April 15, 2020.
58. Damevski K. Expressing measurement units in interfaces for scientific component software. Proceedings of the 2009 Workshop on Component-Based High Performance Computing; 2009:13:1–13:8. https://doi.org/10.1145/1687774.1687787.
59. Ore JP, Detweiler C, Elbaum S. Phriky-Units: a lightweight, annotation-free physical unit inconsistency detection tool. Paper presented at: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. Santa Barbara; 2017:352-355. https://doi.org/10.1145/3092703.3098219.
60. Garny A, Nickerson D, Cooper J, et al. CellML and associated tools and techniques. *Philos Trans Royal Soc A Math Phys Eng Sci*. 2008;366:3017-3043. https://doi.org/10.1098/rsta.2008.0094.

How to cite this article: McKeever S, Bennich-Björkman O, Salah O-A. Unit of measurement libraries, their popularity and suitability. *Softw: Pract Exper*. 2021;51:711–734. https://doi.org/10.1002/spe.2926