The Next 700 Unit of Measurement Checkers

Oscar Bennich-Björkman
Department of Informatics and Media, Uppsala University
Sweden
oscar.bennich-bjorkman@im.uu.se

Steve McKeever
Department of Informatics and Media, Uppsala University
Sweden
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 major, sometimes 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. Instead, scientific code developers often make their own tools or rely on external libraries to help them spot or prevent these mistakes.

We employed a systematic approach to examine and analyse all available physical quantity open-source libraries. Approximately 3700 search results across seven repository hosting sites were condensed into a list of 82 of the most comprehensive and well-developed libraries currently available. In this group, 30 different programming languages are represented. Out of these 82 libraries, 38 have been updated within the last two years. These 38 are summarised in this paper as they are deemed the most relevant.

The conclusion we draw from these results is that there is clearly too much diversity, duplicated efforts, and a lack of code sharing and harmonisation which discourages use and adoption.

CCS Concepts • **Software and its engineering → Software libraries and repositories**; *Language features*; *Software verification and validation*;

Keywords Physical quantities, libraries, units of measurement, open-source, scientific programming.

ACM Reference Format:
Oscar Bennich-Björkman and Steve McKeever. 2018. The Next 700 Unit of Measurement Checkers. In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering (SLE '18), November 5–6, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3276604.3276613

1 Introduction

On the morning of September 23, 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 [43]. 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 physical quantity libraries.

Unit errors can stem both from 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 [37]; 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 built in a 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 use integers with added comments instead.

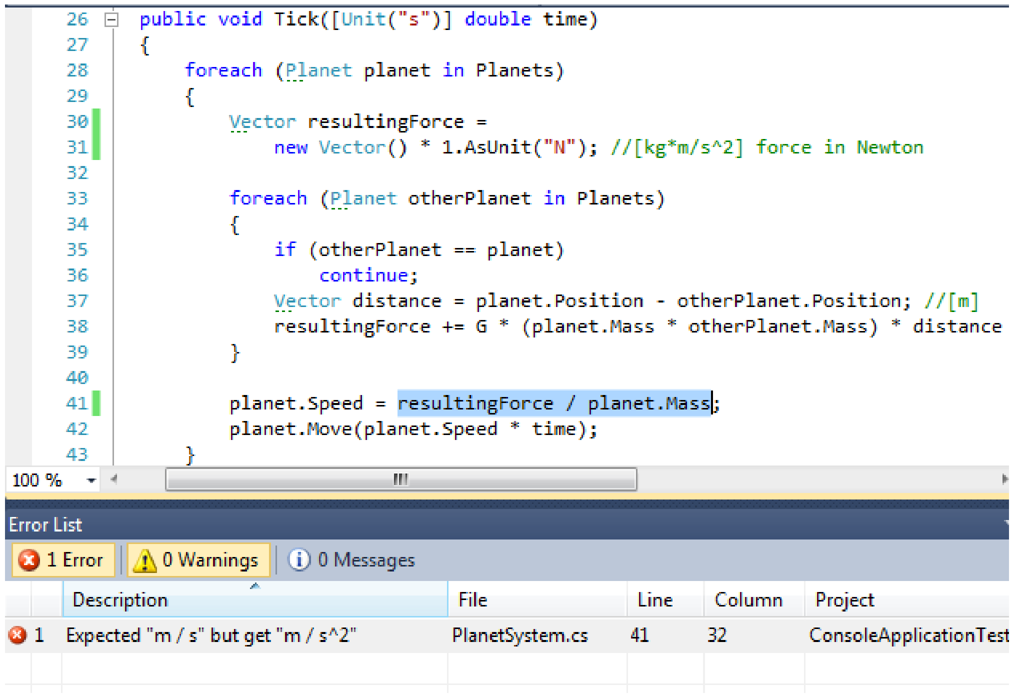
```
int length = 100; // kilometres
int time = 10;    // seconds
```

The second programmer now wants to use these values to calculate speed.

```
int speed = length / time; // m/s
```

The variable `speed` will now have the value of `10`, incorrectly assumed to be `10 m/s`. The issue is that the variable `length` is supposed to represent `10 kilometres`, not `10 metres`. This means that the actual speed is `10000 m/s`, 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. This is in essence the fundamental problem that can lead to such catastrophic events that were described previously.

The only reliable way to solve this is try to remove the human element by having automatic ways to check calculations and make sure they are correct. This type of automatic check is potentially something that could be done at compile



```
26  public void Tick([Unit("s")] double time)
27  {
28      foreach (Planet planet in Planets)
29      {
30          Vector resultingForce =
31              new Vector() * 1.AsUnit("N"); //[kg*m/s^2] force in Newton
32
33          foreach (Planet otherPlanet in Planets)
34          {
35              if (otherPlanet == planet)
36                  continue;
37              Vector distance = planet.Position - otherPlanet.Position; //[m]
38              resultingForce += G * (planet.Mass * otherPlanet.Mass) * distance
39          }
40
41          planet.Speed = resultingForce / planet.Mass;
42          planet.Move(planet.Speed * time);
43      }
```

		Description	File	Line	Column	Project
	1	Expected "m / s" but get "m / s^2"	PlanetSystem.cs	41	32	ConsoleApplicationTest

Figure 1. Compilation error example from Units of Measure Validator for C# [19]

time in a strongly typed language, but unfortunately very few languages support units of measurement currently (see Section 4). Instead, it is up to the software developers to create these checks themselves.

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

```
Kilometre length = 100;
Second time = 10;
Speed speed = length / time; // 10000 m/s
```

Compared to previously, here the compiler now knows exactly what it is dealing with and thus the information that `length` is in kilometres is kept intact and the correct speed 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 formula (which is length over time), he or she uses metres divided by seconds squared. Luckily, the static checker catches this error and informs the programmer.

Unfortunately, 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 existing code base. Libraries like this do already exist, several hundred in fact. The problem thus is not the lack of solutions but the opposite; potential solutions are so numerous that it is hard to get an overview. Moreover, when looking deeper, it quickly becomes apparent that there is no agreed upon standard in the area, 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'.

The goal of this work is to summarise and categorise all open-source libraries that handle physical quantities, comprehensively and systematically. Our study is mainly aimed at two groups. The first group are developers who are in need of a library that handles physical quantities in their preferred language, but do not have the time or resources to find and analyse all relevant alternatives themselves. The second group are those who wish to develop a unit library of their own or contribute to an existing one. In this case, the paper can help them with pointers to what has already been undertaken, categorised and summarised in accessible manner.

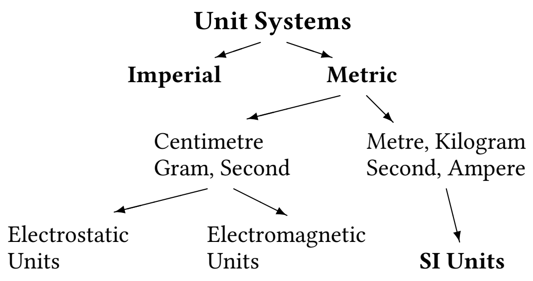


Figure 2. Evolution of units systems

The paper is structured as follows. In the next section we delve into the technical description of units and the various systems that exist. In Section 3 we discuss the low number of languages that have built-in support for units. In Section 4 we briefly describe the history of unit systems and other surveys that cover unit libraries. The bulk of our study is presented in Section 5 where we outline the filtering process applied to open-source libraries to uncover the state-of-play. We summarise the overall results in Section 6, and present exemplars of unit systems for popular languages in Section 7. We conclude our survey in Section 8, and in Section 9 we list some future avenues of research, but stress the need to standardise to promote the use of such libraries.

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" [22].

To explain this further, each quantity is expressed as a number (the magnitude of the quantity) with an associated unit [18]. For example the physical quantity of length can be expressed with the unit metre and the magnitude `10` (`10m`). However, the same length can also be expressed using other units such as centimetres or kilometres, at the same time changing the magnitude (`1000cm` or `0.01km`). Keeping the same physical quantity consistent across multiple units is one of the main functions that the libraries presented in this paper provide, and something that, if not kept in check, can have major consequences.

Physical quantities also come in two types, *base quantities* and *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 [41]. For example length (metre) is a base quantity, and so is time (second). If these two base quantities are combined they express velocity (metre/second or metre $\times$ second^{-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 [38].

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

```
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* [17].

```
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))
```

In concrete syntax this is instead expressed 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 Table 1, therefore many libraries hard code their allowable units. Therefore the inclusion or exclusion of dimensional exponents is one potential way of categorising these libraries.

As was previously mentioned, 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 have equal dimensions [41]. The concept of dimensional analysis is also relevant to the libraries described in this paper, as they often employ similar ways to check for errors. There are also specific software tools that can perform dimensional analysis on existing code (for example the tool described by Cmelik and Gehani [13]).

The examples above were all based on the *International System of Units* (SI) which is the most used and well known unit system, but there exist several other systems in which these physical quantities can be expressed, each with different units for the same quantity. Examples include the *Imperial system*, the *Atomic Units system*, and the *CGS* (centimetre, gram, second) system, see Figure 2. These have evolved over time and branched off from each other.

To understand how these systems relate to units and quantities, think of how to express a certain length in the SI system and the Imperial system, for example 2 metres is about 6.6

Table 1. Some standard derived units and quantities

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

feet. The same quantity (length) is expressed by different units, metres in one system and feet in the other.

Although the SI system is the most used, the Imperial system is still utilised heavily in the United States, which means that keeping track of quantities expressed in different unit systems is important, and is a type of functionality that most unit libraries implement.

3 Existing Language Support

One of the main reasons that 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 [12] with support for units of measurement is Apple's Swift language [7]. The only other well-known language to support units of measurement at the moment is F# [34]. Although compared to Swift, F# is in the bottom 50 in terms of popularity.

In relation to this, it is interesting to note that one thing that Swift and F# have in common is that they are both relatively recently developed programming languages (Swift in 2014 and F# in 2005). The fact that these two languages are the only ones in popular use that support physical quantities would suggest that supporting this functionality is something that needs to be part of the language core, as adding it to an already existing language (such as C# or Python) is either not possible or causes many non-trivial conflicts.

There have also been attempts to provide standard library support for units of measurement in the Java language through the Java Community Process^{SM} Program [39] and the UOMo project [29].

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 [40]. These libraries are all well made, well documented, and viewed by many as a de facto part of the C++ language.

Finally, there are some examples of smaller languages that support physical quantities, such as Nemerle [4] and Frink [21]. However, since these languages are rarely used, the fact that they do have this support is not as relevant as the previous examples.

4 Related Work

Adding units to conventional programming languages has a rich history going back to the 1970s and early 80s, with proposals to extend Fortran [25] and then Pascal [20]. However, these efforts were heavily syntactic and required modifications to the underlying languages. Moreover, the connection with static checking was not understood.

The pioneering foundational work was undertaken by Wand and O'Keefe [44]. They revealed 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. Kennedy extended Wand's work [30] and contributed greatly to the F# project [31]. This strand of work relies on extending the language's static checker. The Osprey type system [28] provides similar functionality for C but with further optimisations to the inference engine. They applied it to mature scientific application code and found hitherto unknown errors.

Representing units in object oriented models can be managed using the Quantity pattern and translated through the Conversion Ratio pattern [23]. This allows users to choose their own unit system for particular values and allows polymorphism to manage the functional diversity required by the assortment of needs [8]. Operating with quantity values and refinement to Java are explored in [33]. However in order to have dimension checking at compile-time, the compiler must be able to manipulate unit expressions. This can be achieved using extended versions of modern languages that support first-class generic types [5].

A more viable starting point for our study would be the work of Hilfinger [26], which showed how to exploit Ada's abstraction facilities, namely operator overloading and type parameterisation, to assign attributes for units of measure to variables and values. When looking more specifically at academic contributions describing unit library usage, the largest is a talk given by Trevor Bekolay from the University of Waterloo at the 2013 SciPy conference. [10].

In his 20-minute 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 physical quantities, going through their functionality, syntax, implementation, and performance. At the end of the talk he also presents a possible "unification of the existing packages that enables a majority of the use cases".

There have also been some attempts at trying to summarise libraries in this area by practitioners, developers working on their own tools. Most of these attempts are limited by the authors looking only 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 slightly beyond simply listing libraries in one language [1, 9, 35, 45].

5 A Systematic Review of Unit Libraries

Our study involved finding, summarising, and categorising unit libraries found on open-source hosting sites. Because there is almost no previous research in this area, the manner in which the study was carried out could not be copied from other researchers but instead had to be designed with this specific goal in mind. Nonetheless, there are similarities to the methodologies employed in other studies that have gathered data from repositories on GitHub such as [11, 15, 36].

5.1 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 found 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 used either 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 [3, 27, 42]. The sites were: GitHub.com, BitBucket.org, GitLab.com, SourceForge.net, CodeProject.com, LaunchPad.net and Savannah.gnu.org.

Even though GitHub is the most extensively used out of these sites and could potentially have sufficed on its own, the intent was for the search to be as comprehensive as possible and therefore the other six sites were also included.

This combination of sites and keywords returned over 65000 total (non-unique) projects, 78 percent 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 30000 results and "Conversion" over 17000. 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 paper (such as conversion of data types).

Because this number of results was impractical to go through one by one, a choice was made to put a limit of a maximum of 200 results (circa 20 pages) per keyword per site. Although the number was chosen arbitrarily, 200 results was seen as a good compromise between thoroughness and feasibility. The relevance of projects outside of the first 200 results would be low and therefore the chance of missing an important contribution is minimal. This cap reduced the number of results from 65000 projects to about 3700, which were then examined manually.

All the sites and search terms are listed in Table 2, 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 limit of 200 was employed and the number in the parenthesis is the total number of search results that would otherwise be available. For example, 200 (17110) for the keyword "Conversion" on GitHub means that 200 results for this keyword were examined but 17110 is the total number of search results found. In the case of 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 added and under that the total number of results that would otherwise be available is shown.

5.2 Analysing Project Validity

After the initial search, all the relevant projects that were found were analysed again. During this second scrutiny, some of these projects that initially looked relevant were deemed to be invalid and were therefore excluded.

A project was deemed invalid if, on closer inspection, it was lacking in one or several areas. These were for example that *the code was incomplete*, *the documentation was severely lacking*, *it was not written in English*, *the code did not work*, or *that the solution was too limited in scope*. Also, 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.

Table 2. 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 (31019)	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 (17110)	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
Amount of projects	1349 (52265)	600 (2651)	301 (323)	692 (4846)	760 (8998)	214 (518)	16 (16)

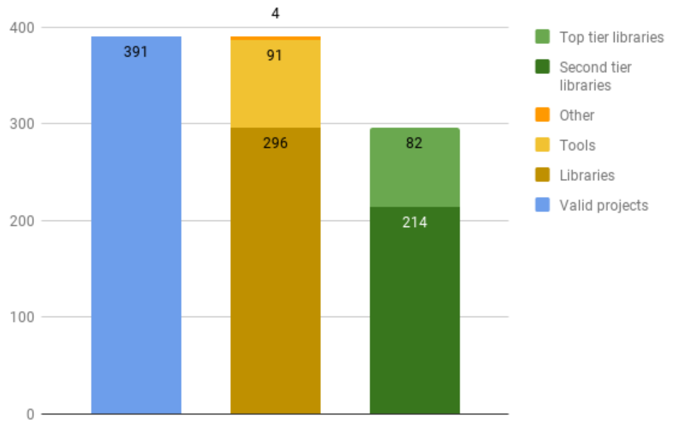


Figure 3. Valid projects, tools, and libraries

5.3 Categorising Top Tier and Second Tier Libraries

In this final step, the libraries that had not been filtered out in the previous steps 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. Because the goal in this part of the process was to categorise 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 is actively being worked on or has recently been updated** - If the library is actively being worked on and updated it generally means that the developers are continuously listening to requests and fixing bugs, resulting in a better and more comprehensive library.
- **The library has a high number of commits** - A library having a high number of commits generally means that it has been worked on for some time, bugs have been fixed, more functionality added, etc. Although the number of commits does not always correlate to quality (as the extent of what a commit is can vary), it is usually a good measure of overall quality. Some libraries have as few as five or ten commits while others have hundreds or even thousands. For the purposes of this paper, a high number of commits is seen as anything above one hundred.
- **The library has a high number of contributors** - Similar to the point above, a high number of contributors can also be a good measure of quality as this means many different people have worked on the project and added things of their own. Even though this is not a perfect measure of quality either, a project with fifty contributors will generally be of higher quality than a project with just one.
- **The library has comprehensive documentation** - Because 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 comprehensiveness of the documentation became an important factor. This is also important for anyone else who wants to use the library, as a lack documentation means that it is harder to use and understand.
- **The library supports 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 has high ratings and is popular** - High ratings (or something similar) is usually a good indication of overall quality. Similarly, a library that many people like and is popular is generally of good quality.

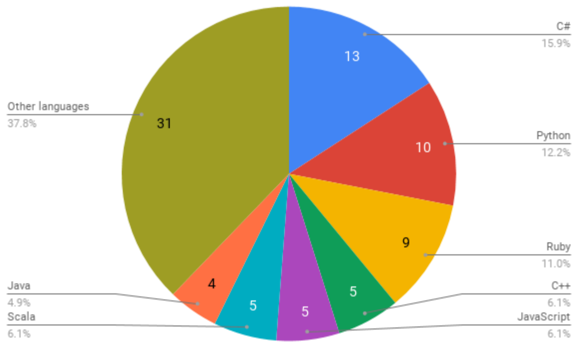


Figure 4. Top tier libraries per language

- **The library has some unique part, is written in an underrepresented language or in some other way sticks out** - Finally, if the library lacks some of the other qualities listed here but instead has a unique way of solving a related problem, provides a unique solution, or is written in an underrepresented language it can also be included in the top tier group. Although novelty is not necessarily a measure of quality we still chose to include some of these libraries as they provide interesting ideas for other developers to look at.

If a library is missing one or several of the criteria that would otherwise qualify it as a top tier library it is instead placed in the second tier group.

6 Overview of 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 91 were tools. From the group of 296 libraries, 214 were deemed to be second tier and 82 to be top tier. See Figure 3.

Of the libraries in the top tier group, approximately 80 percent were hosted on GitHub. Interesting to note is that this is almost identical to the amount of projects that were hosted on GitHub when looking at the group of total search results, which was 78 percent.

For the top tier group the total number of commits was about 36000, an average of about 440 commits per project. The average number of contributors is circa 8.3 per project. But these numbers are heavily influenced by the project Astropy [2] that has almost 22000 commits and 240 contributors alone, making it an extreme outlier in the group. Removing the Astropy data, the total number of commits drops to 14000 and the average commits per project to 175. The average number of contributors per project also goes down to ~5.5.

75 percent of the projects were updated within the last two years (2017 and 2018), and many projects are being actively worked on and updated continuously. Looking at unit support, on average, each library supports about 200 unique units. Regarding programming language distribution in this group, there are 30 different languages in total, with C# being the most popular, closely followed by Python and Ruby (see Figure 4).

It is obvious why compiled languages, like C#, would benefit from unit checking; and why Python which is seen as a scientific scripting language would too. However it is less clear why libraries for Ruby would be so popular.

7 Library Results

In this section the results of the study are presented, split into subsections for each of the chosen languages. The most well-represented languages are presented first. We have also decided to include only libraries that have been updated in 2017 or later in these results. The complete data from the study can be found at our GitHub repository[1]. Each subsection contains information about one of the leading libraries in that particular language, as well as a short code snippet of how the library can be used based on the previous example of time, length, and speed. Naturally there are similarities between languages and libraries but the following code fragments also highlight the variety, and support our thesis that a standard API should be adopted.

7.1 Libraries Written in C#

UnitsNet is one of the best examples of a library written in C#. This library supports a large number of different units, is continuously updated, and worked on by more than 60 developers over many years. The project also has comprehensive documentation explaining the functionality.

UnitsNet relies heavily on the combination of static typing, operator overloading, and extension methods to make unit checking and conversions both simple and the syntax concise. In the example below, all of these three techniques are utilised to create an object of type Speed. Using extension methods, a `Kilometer` object and a `Second` object are effortlessly created with correct values. The objects are then converted through the application of the overloaded division operator, and a compile-time check is made to ensure that the equation is indeed correct.

```
Speed speed = 100.Kilometers() / 10.Seconds() // 10km/s
```

Additionally, conversion between different units can be obtained simply by using the dot notation, such as converting a `metre` to `inches` or an `hour` to `microseconds`.

```
Length meter = Length.FromMeters(1) // 1 meter
double inches = meter.Inches // 39.3701 inches

TimeSpan week = TimeSpan.FromWeeks(1) // 1 week
double hours = week.Hours // 168 hours
```

[1] https://github.com/OscarBennich/Comprehensive-List-Of-Unit-Checker-Libraries

Table 3. Libraries written in C#

Name	Contributors	Commits	Unit Support	Source / URL
CaliperSharp	1	60	130	https://github.com/point85/CaliperSharp
Cubico	1	90	140	https://github.com/irperez/Cubico
Fhir.Metrics	2	53	150	https://github.com/FirelyTeam/Fhir.Metrics
Gu.Units	3	422	50	https://github.com/JohanLarsson/Gu.Units
Quantity System	2	140	300	https://github.com/ibluesun/QuantitySystem
Quantity Types	12	345	300	https://github.com/objorke/QuantityTypes
RedStar.Amounts	1	82	80	https://github.com/petermorlion/RedStar.Amounts
UnitParser	1	25	300	https://www.codeproject.com/Articles/1211504/UnitParser
UnitsNet	61	1760	600	https://github.com/angularsen/UnitsNet

UnitsNet also supports a large number of abbreviations and even localisation support when parsing a unit to a string. See Table 3 for more examples of libraries written in C#.

7.2 Libraries Written in Python

One of the most active libraries for Python is Pint. It supports a large number of units and different unit systems as well as having a large number of contributors. It is updated frequently and is well documented.

Using Pint, one could model the previous example as seen below. First, a `distance` with the magnitude 100.0 `kilometer` is created, subsequently a `time` quantity of 10.0 seconds is created. These two quantities are then combined to create a third quantity, speed, by dividing the distance with the time. This produces the expected result of a quantity with the magnitude 10000 and the unit `meter / second`

```
>>> distance = 100.0 * ureg.kilometer
>>> time = 10.0 * ureg.second

>>> speed = distance / time
>>> print(speed)
10 kilometer / second
```

The variable speed can be converted to other meaningful quantities.

```
>>> speed.to(ureg.meter / ureg.second )
<Quantity(10000.0, 'meter / second')>
```

See Table 4 for more examples of libraries written in Python.

7.3 Libraries Written in Ruby

A representative example of a library written in Ruby is Ruby Units. In Ruby Units each quantity is defined using the `Unit` object and this class handles all conversions between different units.

Like the examples from previous libraries, it is easy to model distance, time, and speed using Ruby Units. And similar to UnitsNet it can also be conveyed in one line.

```
distance  = Unit.new("100 km")
time = Unit.new("10 seconds")
speed = distance / time
```

```
speed = Unit.new("100 km / 10 seconds")
```

Ruby Units also allows for the creation of complex units using dimensional exponents.

```
newton = Unit.new("1 kg m s^-2")
watt = Unit.new("1 kg m^2 s^-3")
```

See Table 5 for more examples of libraries written in Ruby.

7.4 Libraries Written in C++

The leading library for C++ is BoostUnits. BoostUnits 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) and can be seen as the de facto unit library in C++. 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 [5]. Our example can be encoded as follows, note that one must explicitly convert kilometers into the base unit of length which is meters.

```
quantity<si::length,int>
        length (100 * si::kilometers);
quantity<si::seconds,int> time = 10;

quantity<si::velocity,int>
        speed = length / time;

// speed = 10000 m s^-1
```

While the library attempts to make simple dimensional computations easy to code, it is not tied to any particular unit system, such as SI or CGS. It provides a highly flexible compile-time system for dimensional analysis, supporting arbitrary collections of base dimensions, and rational powers of units so users can define their own systems. It accomplishes all of this via template meta-programming techniques. With modern optimizing compilers, this results in zero run-time overhead for quantity computations relative to the same code without unit checking. See Table 6 for more examples of libraries written in C++.

Table 4. Libraries written in Python

Name	Contributors	Commits	Unit Support	Source / URL
Aegon	2	21	150	https://github.com/lukaskollmer/aegon
Astropy	267	22598	150	https://github.com/astropy/astropy
Misu	3	162	450	https://github.com/cjrh/misu
Pint	83	1004	300	https://github.com/hgrecco/pint
Pyvalem	2	48	70	https://github.com/xnx/pyqn
Quantiphy	1	284	40	https://github.com/KenKundert/quantiphy
Quantities	18	461	300	https://github.com/python-quantities/python-quantities
Scimath	13	313	317	https://github.com/enthought/scimath

Table 5. Libraries written in Ruby

Name	Contributors	Commits	Unit Support	Source / URL
Alchemist	12	150	500	https://github.com/halogenandtoast/alchemist
Phys-Units	1	123	2300	https://github.com/masa16/phys-units
Ruby Units	25	425	150	https://github.com/olbrich/ruby-units
Unit_Soup	1	26	0	https://github.com/rutvij47/unit_soup
Unitwise	4	257	300	https://github.com/joshwlewis/unitwise

Table 6. Libraries written in C++

Name	Contributors	Commits	Unit Support	Source / URL
BoostUnits	640	24	200	https://github.com/boostorg/units/
PhysUnits	4	75	100	https://github.com/martinmoene/PhysUnits-CT-Cpp11
Units	12	371	150	https://github.com/nholthaus/units
Units	1	48	30	https://github.com/tonypilz/units
Units and Measures	1	20	50	https://www.codeproject.com/Articles/1088293/ Units-and-measures-for-Cplusplus

7.5 Libraries Written in JavaScript

One of the most mature library written in JavaScript is JS-Quantities. JS-Quantities is actively developed, supports many units and has good documentation. It is also interesting to note that JS-Quantities originally was a JavaScript port of the Ruby Units library (see Section 7.3). Therefore it shares much of the same functionality. JS-Quantities also defines several distinct alias for each unit it supports so there is considerable flexibility as to how a user can create a new quantity object. Below, three distinct ways of creating a new quantity for Speed are shown. Similar to Ruby Units a unit can be derived using dimensional exponents, shown in the third example.

```
speed = Qty('100 km/10s');
speed = Qty('100 kilometres/10 seconds');
speed = Qty('10000 m s-1');
```

See Table 7 for more examples of libraries written in JavaScript.

7.6 Libraries Written in Scala

There are generally fewer Scala libraries but one elegant example is Squants. It supports a fair number of units as well as providing very extensive documentation, it is also being actively supported by its contributors.

The code below shows how speed can be created *on the fly* in one line from kilometers and seconds, similar to several other libraries. Note however that although speed is created using kilometers as the unit of length, the default unit for the resulting quantity speed is still meter per second or `m/s`.

```
scala> val speed = 100.kilometers / 10.seconds
 speed: squants.motion.Velocity = 10000.0 m/s
```

One interesting function that Squants provides, that most other libraries do not, is the ability to make an approximation for a specific unit. This results in it being possible to define that any magnitude for, say, a meter should be seen as accurate if it is within x meters of the actual value (0.5 meters in this case).

```
scala> implicit val tolerance = Meters(0.5)
```

Table 7. Libraries written in JavaScript

Name	Contributors	Commits	Unit Support	Source / URL
Flowsnake	1	167	650	https://github.com/Eldelshell/flowsnake
JS-Quantities	21	329	150	https://github.com/gentooboontoo/js-quantities
Mezur	1	32	100	https://github.com/guyisra/mezur
Units	1	194	50	https://github.com/stak-digital/units
Unitz	2	20	70	https://github.com/ClickerMonkey/unitz

Table 8. Libraries written in Scala

Name	Contributors	Commits	Unit Support	Source / URL
Coulomb	2	311	70	https://github.com/erikerlandson/coulomb
SI	1	57	50	https://gitlab.com/h2b/SI
Squants	28	432	80	https://github.com/typelevel/squants

Table 9. Libraries written in Java

Name	Contributors	Commits	Unit Support	Source / URL
Caliper	1	139	80	https://github.com/point85/caliper
SI-Units	7	274	120	https://github.com/unitsofmeasurement/si-units
JSR 385	10	579	30	https://github.com/unitsofmeasurement/unit-api

Once this tolerance level is set the approximation check can be utilised in four different ways (=~, ~=, ≈, or the method approx).

```
scala> val distanceOne = Meters(2.0)
scala> val distanceTwo = Meters(1.5)
scala> distanceOne approx distanceTwo // true
```

See Table 8 for more examples of libraries written in Scala.

7.7 Libraries Written in Java

One of the better examples of a library written in Java is Caliper. Caliper supports many units, is actively developed and provides good documentation. In the example below the two unit objects km and sec are used to keep track of their respective quantities. Although creating quantities using Caliper comes with a great degree of flexibility, the syntax is generally more verbose than that of other libraries shown in this paper.

```
UnitOfMeasure km =
       sys.getUOM(Prefix.KILO, Unit.METRE);
UnitOfMeasure sec = sys.getSecond();

Quantity kilometre = new Quantity(100d,km);
Quantity second = new Quantity(10d,sec);

// speed is 10 km/sec
Quantity speed = kilometre.divide(second);
```

Caliper also supports dynamic unit conversion in a rather elegant manner.

```
UnitOfMeasure m = sys.getUOM(Unit.METRE);
UnitOfMeasure cm = sys.getUOM(Prefix.CENTI, m);

Quantity q1 = new Quantity(2d, m);
Quantity q2 = new Quantity(2d, cm);

// add two quantities.  q3 is 2.02 metre
Quantity q3 = q1.add(q2);
```

Moreover, similar to UnitsNet, conversion between different units can also be undertaken explicitly using the dot notation. See Table 9 for more examples of libraries written in Java.

8 Discussion

What is the reason behind the fact that there are so many libraries and why seemingly so few of them are used? When it comes to the first part of this question we believe Bekolay put it quite well when he said that *"making a physical quantity library is easy but making a good one is hard"* [10]. The core issue that these libraries aim to solve is something that is relatively simple to understand, familiar, as well as easy to create a simple solution for. This makes it well-suited as a hobby project or a programming assignment.

Implementation difficulties arise when trying to make a more complete library, including more units, more operators, or more complex functionality overall. At this point it generally requires too much effort for those undertaking this type of task as a novice or for fun. This seems to be a viable contributing factor as to why there are so many projects and so few truly comprehensive ones. In addition, there is quite a sharp decline in quality when looking at projects outside of the top tier group, which supports this hypothesis.

But even in the cases of libraries with extensive scope and complexity, the lack of cooperation is apparent from the clear absence of references to other projects or research in the area. As was mentioned previously there exists a few examples of such references but even in these cases they are superficial.

A fundamental reason why unit checking of any sort is not common in scientific or engineering software development is that it is not mandatory, nor does the compiler require it. Many projects start small, using prototyping or test-driven approaches, within a clearly acknowledged unit usage context and it is only when the project scales up that problems start to arise.

Damevski [16] argues that scientific programmers should not be burdened by units at each statement in their programs, but that units should be inserted in software component interfaces. Anecdotal evidence to support this claim stems from the physics and chemistry worlds when they discuss energy conversions, for instance. While the physicist works in electron volts per formula unit, the chemist thinks in terms of kilojoules per mole. In such cases, not only are the units different but so are the magnitudes. A component interface based discipline means that the consequences of local unit mistakes are underestimated. Developers might feel that it is not worth going through the effort of using one of the available libraries, yet when they come to combining their codes with others the conversions need to be explicitly introduced. This is compounded by the general problem of accuracy that most libraries are unable to deal with, and manifests itself when trying to add quantities of vastly different magnitude to each other, such as a light year and an inch. One example of a project that specifically aims to deal with this accuracy problem is UnitParser [24]. According to the documentation for this library it is able to support numbers as big as $10^{2147483647}$ as well as over 27-digit precision. This is achieved through the implementation of a "mixed system" in the underlying unit class.

A reason for not using many available tools is that they introduce subtle inefficiencies in how they do unit conversions. In the case of assignment expressions, some methods simply convert the operands of operators such as + to use the units of the leftmost operand. While this is the most straightforward to implement, it is not always the most sensible or efficient [14]. Even light weight run-time conversions could slow computations down, and when typical scientific applications require upwards of 100,000 computation hours per month, the impact might not be so negligible.

9 Conclusion and Future Directions

The results presented in this paper are intended to expose the large but, seemingly, little known area of unit checker libraries. We presented 38 open-source physical quantity libraries. This group represents the most comprehensive and actively-developed libraries, systematically chosen from approximately 3700 search results across seven sites, and involving 30 different programming languages. On such a scale, this type of study is the first of its kind.

We have confined our study to mainstream languages but this study should be expanded to include popular spreadsheets [6], mathematical tools and domain specific scientific notations, in which there is plenty of scope to successfully leverage unit checking [14] before code generation.

The software engineering benefits of adopting unit checking and automatic conversion support is unequivocal. Programmers make mistakes, every line of code can be subjected to a large range of values and dependencies. Providing unit support is an obvious way to mitigate against such human error. Unit systems help manage complexity, allowing the programmer to rely on the checker to ensure correctness and not on themselves or their colleagues. Moreover; requirements of systems change over time, and so do programming teams, ensuring the code maintains the original unit requirements informally is careless.

We need a better understanding of the reasons for the lack of adoption. It now seems pertinent to question developers, who have an assortment of needs and who could benefit most from such libraries, in order to ensure their wishes are understood and a clear set of requirements emerges.

Efforts should be focused on developing a language neutral interface that models the core requirements of the community and allows programmers to manage units in an indistinguishable language agnostic fashion. Similar in spirit to what Landin [32] was proposing for programming languages back in the mid 1960s.

References

[1] 2012. Unit Conversion Shootout. Retrieved July 2nd, 2018 from https://github.com/JustinLove/unit_conversion_shootout

[2] 2018. A common core package for Astronomy in Python. Retrieved July 2nd, 2018 from https://github.com/astropy/astropy

[3] 2018. Comparison of source code hosting facilities. Retrieved July 2nd, 2018 from https://en.wikipedia.org/wiki/Comparison_of_source_code_hosting_facilities

[4] 2018. Nemerle Programming Language. Retrieved July 2nd, 2018 from http://nemerle.org/

[5] Eric Allen, David Chase, Victor Luchangco, Jan-Willem Maessen, and Guy L. Steele, Jr. 2004. Object-oriented Units of Measurement. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '04)*. ACM, New York, NY, USA, 384–403. https://doi.org/10.1145/1028976.1029008

[6] Tudor Antoniu, Paul A. Steckler, Shriram Krishnamurthi, Erich Neuwirth, and Matthias Felleisen. 2004. Validating the Unit Correctness of Spreadsheet Programs. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*. IEEE Computer Society, Washington, DC, USA, 439–448. http://dl.acm.org/citation.cfm?id=998675.999448

[7] Apple. 2018. Swift Open Source. Retrieved July 2nd, 2018 from https://swift.org

[8] Benoit Baudry and Martin Monperrus. 2015. The Multiple Facets of Software Diversity: Recent Developments in Year 2000 and Beyond. *ACM Comput. Surv.* 48, 1, Article 16 (Sept. 2015), 26 pages. https://doi.org/10.1145/2807593

[9] Tim Baumann. 2017. Quantities, Libraries in other programming languages. Retrieved July 2nd, 2018 from https://github.com/timjb/quantities/wiki/Links

[10] Trevor Bekolay. 2013. A comprehensive look at representing physical quantities in Python. Video. In *Scientific Computing with Python.* Retrieved July 2nd, 2018 from https://conference.scipy.org/scipy2013/presentation_detail.php?id=174

[11] Tegawendé F. Bissyandé, Ferdian Thung, David Lo, Lingxiao Jiang, and Laurent Réveillère. 2013. Popularity, Interoperability, and Impact of Programming Languages in 100,000 Open Source Projects. *2013 IEEE 37th Annual Computer Software and Applications Conference* (2013), 303–312.

[12] TIOBE Software BV. 2018. TIOBE Index for June 2018. Retrieved July 2nd, 2018 from https://www.tiobe.com/tiobe-index/

[13] Robert F. Cmelik and Narain H. Gehani. 1988. Dimensional Analysis with C++. *IEEE Softw.* 5, 3 (May 1988), 21–27. https://doi.org/10.1109/52.2021

[14] Jonathan Cooper and Steve McKeever. 2008. A model-driven approach to automatic conversion of physical units. *Softw. Pract. Exper.* 38, 4 (2008), 337–359. https://doi.org/10.1002/spe.828

[15] Valerio Cosentino, Javier Luis Cánovas Izquierdo, and Jordi Cabot. 2017. A Systematic Mapping Study of Software Development With GitHub. *IEEE Access* 5 (2017), 7173–7192. https://doi.org/10.1109/ACCESS.2017.2682323

[16] Kostadin Damevski. 2009. Expressing Measurement Units in Interfaces for Scientific Component Software. In *Proceedings of the 2009 Workshop on Component-Based High Performance Computing (CBHPC '09).* ACM, New York, NY, USA, Article 13, 8 pages. https://doi.org/10.1145/1687774.1687787

[17] Bureau International des Poids et Mesures. 2014. SI Brochure: The International System of Units (SI), 8th Edition, Dimensions of Quantities. Retrieved July 2nd, 2018 from https://www.bipm.org/en/publications/si-brochure/chapter1.html

[18] Bureau International des Poids et Mesures. 2014. SI Brochure: The International System of Units (SI), 8th Edition, Quantities and Units. Retrieved July 2nd, 2018 from https://www.bipm.org/en/publications/si-brochure/chapter1.html

[19] Henning Dieterichs. 2012. Units of Measure Validator for C#. Code Project. Retrieved July 2nd, 2018 from https://www.codeproject.com/Articles/413750/Units-of-Measure-Validator-for-Csharp

[20] A Dreiheller, B Mohr, and M Moerschbacher. 1986. Programming Pascal with Physical Units. *SIGPLAN Not.* 21, 12 (Dec. 1986), 114–123. https://doi.org/10.1145/15042.15048

[21] Alan Eliasen. 2018. Frink Programming Language. Retrieved July 2nd, 2018 from https://frinklang.org

[22] Joint Committee for Guides in Metrology (JCGM). 2012. International Vocabulary of Metrology, Basic and General Concepts and Associated Terms (VIM). Retrieved July 2nd, 2018 from https://www.bipm.org/utils/common/documents/jcgm/JCGM_200_2012.pdf

[23] Martin Fowler. 1997. *Analysis Patterns: Reusable Objects Models.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[24] Alvaro Carballo Garcia. 2018. UnitParser. Retrieved July 2nd, 2018 from https://www.codeproject.com/Articles/1211504/UnitParser

[25] Narain Gehani. 1977. Units of measure as a data attribute. *Computer Languages* 2, 3 (1977), 93 – 111. https://doi.org/10.1016/0096-0551(77)90010-8

[26] Paul N. Hilfinger. 1988. An Ada Package for Dimensional Analysis. *ACM Trans. Program. Lang. Syst.* 10, 2 (April 1988), 189–203. https://doi.org/10.1145/42190.42346

[27] Neil Chue Hong. 2018. Choosing a repository for your software project. Retrieved July 2nd, 2018 from https://software.ac.uk/resources/guides/choosing-repository-your-software-project

[28] Lingxiao Jiang and Zhendong Su. 2006. Osprey: A Practical Type System for Validating Dimensional Unit Correctness of C Programs. In *Proceedings of the 28th International Conference on Software Engineering (ICSE '06).* ACM, New York, NY, USA, 262–271. https://doi.org/10.1145/1134285.1134323

[29] Werner Keil. 2018. Eclipse UOMo Project. Retrieved July 2nd, 2018 from http://www.eclipse.org/uomo/

[30] Andrew Kennedy. 1994. Dimension Types. In *Programming Languages and Systems—ESOP'94, 5th European Symposium on Programming*, Donald Sannella (Ed.), Vol. 788. Springer, Edinburgh, U.K., 348–362.

[31] Andrew Kennedy. 2009. Types for Units-of-Measure: Theory and Practice. In *Central European Functional Programming School - Third Summer School, CEFP 2009, Budapest, Hungary, May 21-23, 2009 and Komárno, Slovakia, May 25-30, 2009, Revised Selected Lectures.* 268–305. https://doi.org/10.1007/978-3-642-17685-2_8

[32] P. J. Landin. 1966. The Next 700 Programming Languages. *Commun. ACM* 9, 3 (March 1966), 157–166. https://doi.org/10.1145/365230.365257

[33] Tanja Mayerhofer, Manuel Wimmer, and Antonio Vallecillo. 2016. Adding Uncertainty and Units to Quantity Types in Software Models. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering (SLE 2016).* ACM, New York, NY, USA, 118–131. https://doi.org/10.1145/2997364.2997376

[34] Microsoft. 2018. F# Software Foundation. Retrieved July 2nd, 2018 from https://fsharp.org

[35] Martin Moene. 2013. Physical Units. Retrieved July 2nd, 2018 from https://github.com/martinmoene/PhysUnits-CT

[36] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating GitHub for engineered software projects. 22 (April 2017). https://doi.org/10.1007/s10664-017-9512-6

[37] Peter G. Neumann. 1992. Illustrative Risks to the Public in the Use of Computer Systems and Related Technology. *SIGSOFT Softw. Eng. Notes* 17, 1 (Jan. 1992), 23–32. https://doi.org/10.1145/134292.134293

[38] The National Institute of Standards and Technology. 2015. International System of Units (SI): Base and Derived. Retrieved July 2nd, 2018 from https://physics.nist.gov/cuu/Units/units.html

[39] Java Community Process. 2016. JSR 363: Units of Measurement API. Retrieved July 2nd, 2018 from https://jcp.org/en/jsr/detail?id=363

[40] Matthias Schabel and Steven Watanabe. 2017. Boost C++ Libraries, Chapter 43 (Boost.Units 1.1.0). Retrieved July 2nd, 2018 from ttps://www.boost.org/doc/libs/1_65_0/doc/html/boost_units.html

[41] Ain A Sonin. 2001. *The Physical Basis of Dimensional analysis.* Technical Report. Massachusetts Institute of Technology. Retrieved July 2nd, 2018 from http://web.mit.edu/2.25/www/pdf/DA_unified.pdf

[42] Stackify. 2017. Top Source Code Repository Hosts: 50 Repo Hosts for Team Collaboration, Open Source, and More. Retrieved July 2nd, 2018 from https://stackify.com/source-code-repository-hosts/

[43] Arthur Stephenson, Lia LaPiana, Daniel Mulville, Frank Bauer Peter Rutledge, David Folta, Greg Dukeman, Robert Sackheim, and Peter Norvig. 1999. Mars Climate Orbiter Mishap Investigation Board Phase I Report. Retrieved July 2nd, 2018 from https://llis.nasa.gov/llis_lib/pdf/1009464main1_0641-mr.pdf

[44] Mitchell Wand and Patrick O'Keefe. 1991. Automatic Dimensional Inference. In *Computational Logic - Essays in Honor of Alan Robinson.* 479–483. citeseer.ist.psu.edu/wand91automatic.html

[45] Stefano Zaghi. 2017. Fury Library Comparison. Retrieved July 2nd, 2018 from https://github.com/szaghi/FURY#libraries