Understanding Reasons Behind the Lack of Adoption of Units of Measure (UoM) Libraries

Omar-Alfred Salah

Subject: Information Systems
Corresponds to: 30 hp

supervised by: Steve McKeever

VT 2019
Abstract

The concepts of units and units of measurement (UoM) are used in scientific and mathematical applications to encode variables and their types with units. There are many UoM libraries, as identified by previous research, that exist that can handle and prevent unit errors and failures that can arise if units are not handled properly, otherwise, the consequences of not using UoM libraries may have possibly disastrous implications. Previous research has categorised thousands of UoM libraries, an indication that the 'wheel is being reinvented' time and time again instead of building on what was already done, indicating that there exists a lack of adoption of these UoM libraries. It is therefore important to understand the scale of the lack of adoption and why that is the case. An exploratory-DSR style research was employed and developers and scientists were surveyed and interviewed to inquire why this is the case, with results ranging from unawareness of these UoM libraries, to specific performance concerns and even tradition and sticking to what already works. Towards the end, the thesis suggests possible solutions to relieving the lack of adoption of these UoM libraries as recommendation for further UoM solutions, with recommendations ranging from including UoM libraries in standard libraries of programming languages to reducing the complexity and verbosity of UoM solutions.

**Keywords**— physical quantities, libraries, units of measure, UoM, lack of adoption
# Contents

1 Introduction ......................................................... 4
   1.1 Background .................................................. 4
   1.2 Problem Description ........................................ 4
   1.3 Existing Research and Contribution ....................... 5
   1.4 Research Goal ............................................... 7
   1.5 Delimitations ............................................... 7

2 Theory ...................................................................... 8
   2.1 Units, Quantities and Dimensions ......................... 8
   2.2 Base, Derived Units and Representation ................... 8
   2.3 Unit Equivalence and Equality ............................... 11
   2.4 Dimensional Analysis, Consistency and Unit Conversion Methods ......................................................... 12
      2.4.1 Language Support and Libraries ....................... 15
      2.4.2 Possible Reasons for General Lack of Adoption ......................................................... 15
      2.4.3 JSR-385 Programming ................................... 16

3 Methodology .......................................................... 18
   3.1 Methodological Approach ...................................... 18
   3.2 Methods of Data Collection ..................................... 19
      3.2.1 Formulation of Questionnaire ......................... 20
      3.2.2 Formulation of the Interview Template ............... 23
      3.2.3 Forum and Bulletin Board Discussions ............... 25

4 Results ..................................................................... 26
   4.1 Web Questionnaire ............................................... 26
      4.1.1 Role-Adopter (Y) Results ............................... 28
      4.1.2 Role-Nonadopter (N) Results ........................... 34
   4.2 Interview Material ............................................... 42
   4.3 Forum and Bulletin boards ..................................... 42

5 Discussion and Analysis ............................................. 43
1 Introduction

1.1 Background

Many papers on this matter begin with the failure of the Mars orbiter which went missing while attempting to enter the Mars atmosphere. Part of the software in the orbiter was written and off sourced to Lockheed Martin, which did not comply with NASA’s Software Interface Specification (SIS). The piece of software used to calculate trajectories then used these results – expected to be in newton seconds – to update the predicted position of the spacecraft. The orbiter was either destroyed in the atmosphere or re-entered heliocentric space after leaving Mars’ atmosphere.

In addition to the Mars orbiter, there are many other unit mishaps. In 1984, the space shuttle “Discovery STS-18” ended up being positioned upside down because the engineers had mistaken feet for miles [1]. Also in 2003, a roller coaster ride in Tokyo broke down, the error arose due to the use of metric units in the design papers for the ride [2]. Taking these mishaps into consideration, it is very possible that many more blunders have gone unreported in both non safety and safety critical systems. Therefore, ignoring units of measurements and the role that they play in code is proven to be short sighted, potentially leading to disasters that could cost a lot of money depending on the scale of the field or area.

1.2 Problem Description

Although an insulated academic scientific group/community might adhere to the same unit measurement standard based on specific guidelines within their communities, the problem becomes evident as different scientific communities and academic circles try to integrate their systems since they might use different units, especially if different system of measurements are used, such as the Imperial or Metric system. If one were to take for example, the equation $F = ma$ in a uncommented codebase (and discounting for the SI system), the mass variable could mean different things for developers and scientists with different backgrounds, shall it be in pounds, kilograms or stones? This decision has a very large effect on the value of $F$, allowing $F$ to take potentially vastly different values. This could lead to a ripple effect with disastrous consequences as $F$ would be used elsewhere in the codebase.
Moreover, the lack of an authoritative estimate of how frequently unit inconsistencies occur or their cost acts as a central figure to the fuzziness and lack of certainty to this issue. Therefore, it is possible that the absence of this missing figure might lead researchers, scientists and academics to not take these unit inconsistency errors with enough initial seriousness, that is, until the time comes when their systems need to integrate, with incompatible unit measurements and specifications.

1.3 Existing Research and Contribution

It is important to study this now as patterns that have previously emerged (such as the Quantity pattern, as proposed by Fowler [3]) exist, yet this type of problem is still plaguing system developers and has been for a long time (since possibly the inception of programming). Unit conversion errors still exist and are not handled in the most effective manner. The ubiquitousness of unit measurement libraries (numbering over thousands of projects on GitHub) [4] with no commonly held standard highlights a growing concern in this area. The fact that so many libraries exist is "reinventing the wheel" done countless amounts of time, but despite this, no standard exists so that developers may effectively combine their codebases together without having to modify them. Therefore, identifying the gravity of this issue becomes pertinent as figures produced from this issue would act as an impetus for developers to take the issue into consideration, reduce system integration time and thereby, costs.

Specifically, there is no research done as to how prevalent the issue of unit conversion errors is in academic circles, but rather, frequently cited disasters such as the Mars orbiter are always referred to when bringing this topic up for discussion. Moreover, a figure like this is not even prevalent in the business world where possible unit conversion errors in financial applications and currency converters might exist [5]. The implications of this even extend to the medical field where it is possible that unit conversion errors may arise when e.g: administering doses of medicine using machines as mentioned in [6]. Therefore, the intended contribution of this work will be to highlight the severity of the issue and to investigate why commonly available libraries are not used (in favour of custom solutions, or no solution at all).

Regarding existing research in the field, a lot of research has been carried out on methods to handle unit conversion errors in systems. Papers of interest to this thesis range from defining the definition of a unit itself, algorithms and methods to handle isolate, locate and solve unit
conversion errors, and/or verify the dimensional correctness of a program using dimensional analysis methods. A lot of work has been committed to the definition of units, dimensional analysis, and algorithms and methods to solve unit conversion errors. For example: Krisper et al.’s [7] approach on creating an improved version of the Quantity pattern based on Fowler’s [3] approach in his book on Analysis patterns. Another key piece of work by Rosu [8] includes setting standards for safety policy checking and general design principles for measurement unit tools so that the problem can be potentially avoided at initial stages of system construction. More work on this subject includes different approaches in expressing and modelling measurement units [9].

This has been brought up because so many methods, tools and libraries exist to compute dimensional inference and provide unit safety in code, yet UoM libraries are being reinvented time and time again, is this because existing UoM libraries are lacking in terms of how they infer their units and their corresponding dimensions? Or is it because scientists and developers do not know better and are not aware of the existence of these methods? These rhetorical questions provide compelling reasons to survey and interview scientists and developers in order to better understand their needs and understand the phenomena behind the reinvention and general lack of adoption of these unit libraries. The knowledge produced from answering these questions may better the design of further UoM solutions as UoM library developers understand the pitfalls and weaknesses of their creations. This is a cornerstone of DSR (design science research) which will be discussed more in the methodology section. Nevertheless, the intended outcome of this thesis is to revaluate the state of UoM libraries in practice today and provide knowledge for future UoM artefacts.
1.4 Research Goal

A large issue that stands out is the lack of adoption of so-called Units of Measure (UoM) libraries as identified by previous research. Therefore, a key cornerstone for this master’s thesis is the need to better understand the reasons for the lack of adoption of these UoM libraries and ensure that their needs for new requirements of unit measurement libraries are properly identified and understood (in the event that UoM libraries do not meet the requirements of developers, scientists, and researchers). Therefore, the main idea is to interview, question, and survey scientists and developers to understand reasons behind the lack of adoption so that a set of requirements may emerge that may improve UoM. A better understanding of the frequency of these errors would highlight the importance of proper and handled unit conversions. All in all, the goals of this research are summarised below:

1. What are the reasons to the lack of adoption of UoM libraries?
2. What is the severity of this issue in terms of frequency and cost?

1.5 Delimitations

To begin with, there are three delimitations to this study. The first delimitation is that only UoM libraries will be looked at, this means any other tool not related to units is irrelevant to the study. This is done to limit the scope to just UoM libraries and the issues surrounding them. In addition to that, the second delimitation when collecting data using data collection methods is to gather the opinions of anyone that has worked with units and units of measurement or a related concept so that the data gathered is more refined and related to issues surrounding units of measurement in software. The last delimitation is that this thesis strictly focuses exploring the reasons to the lack of adoption of UoM libraries in order to produce the necessary knowledge that experts of this discipline can use to design solutions for their field problems, anything else is merely secondary to the research.
2 Theory

Since the thesis aims to explore the reasons to the lack of adoption of unit measurement libraries and place a figure on the severity of the issue. It would thus be vital to define key fundamental concepts that will be of importance as the usage of these terms would occur quite often in the thesis. Therefore, these concepts will be briefly explained in the following sub-sections.

2.1 Units, Quantities and Dimensions

In simple terms, a unit of measurement is a standardised quantity of a physical property. To take an example, 5m indicates that the metre is used as a unit of length that represents a predetermined length, times 5, with the number 5 expressed as the magnitude of the quantity. If one were to use the same magnitude (5) for a different unit of the same base quantity or dimension (length, but using the micrometer or the millimeter), then the physical quantity of the length would change. Since there are many types of predetermined quantities of measurements, the fact that many units of measurements exist becomes a natural occurrence not out of the ordinary. This essentially is the foundation of many of the problems that would occur in software systems since physical quantities can take on different forms depending on the representation i.e: unit of measurement, used. Despite this, because of the importance of a set of well defined, easily accessible and agreed on measurements used ubiquitously in today in society, units should be chosen with care so that the representation of a physical property is recognisable and easy to realise to a high degree of accuracy.

2.2 Base, Derived Units and Representation

One very important set of units is the Le Systeme International d’Unites (SI) [10], the internationally used and recognisable set that was hinted at previously. The SI standard is based on 7 physically independent dimensions/base quantities: mass, length, time, electric current, thermodynamic temperature, amount of substance, and luminous intensity. Each of these dimensions have corresponding base units: kilogram, meter, second, ampere, kelvin, mole and candela. Different dimensions are also recognized by the SI system such as meter per second squared (acceleration) or newtons (kilogram times meter per second squared) to measure force. Commonly used units such as hours or minutes lie outside the SI system, but
are accepted for use alongside the SI. Consequently, these base quantities can be combined together to form *derived* quantities, such as the velocity of a moving object in $ms^{-1}$ or further the acceleration of the object, which is a derivation and the literal derivative of the velocity in $ms^{-2}$. To ensure correctness of units in equations, *dimensional analysis* must be used. So using the previous example, if one were to divide length by time in seconds, then the result must be in $ms^{-1}$. The existence of e.g: the Imperial system or the English system, each with their own sets of units increases the amount of number of units that can be represented. A reference on SI units can be found in the SI brochure [10] as well as the NIST reference on constants, units and uncertainties [11].

<table>
<thead>
<tr>
<th>Name</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>metre</td>
<td>m</td>
</tr>
<tr>
<td>kilogram</td>
<td>kg</td>
</tr>
<tr>
<td>second</td>
<td>s</td>
</tr>
<tr>
<td>ampere</td>
<td>A</td>
</tr>
<tr>
<td>kelvin</td>
<td>K</td>
</tr>
<tr>
<td>mole</td>
<td>mol</td>
</tr>
<tr>
<td>candela</td>
<td>cd</td>
</tr>
</tbody>
</table>

Since the derived units are a specific combination of base units, they sometimes have their own name for simplicity or historic purposes [11]. Examples are most commonly represented in numerator/denominator form such as: Force in Newtons ($kgms^{-2} = N$), Power in Watt ($kgm^2s^{-3} = W$). In addition to that, they can also be represented in the form of a 7-tuple list (where $v =$ magnitude and $[Q]$ the measurement unit to be derived from the 7-tuple list), $v [Q] = v [ [m], [kg], [s], [A], [K], [mol], [cd] ]$. Instantiation of the tuple list is shown below in the following examples:

- $70 \text{ kg} = 70 [ 0, 1, 0, 0, 0, 0, 0 ]$
- $5 \text{ Nm} = 5kgm^2/s^2 = 5 [ 1, 2, -2, 0, 0, 0 ]$
- $10 \text{ m/s}^2 = 10 [ 1, 0, -2, 0, 0, 0, 0 ]$
From the list above, it is simple to derive units from base units as they take different forms in the tuple, even a base unit can also be expressed in 7-tuple form. The new unit of measurement is derived by counting the number of base units and combining it to form a new unit of measurement. Example: in the third list, the unit of measurement for acceleration was derived by setting the first element of the tuple (metres) to 1 and the third element of the tuple (seconds) to -2, which indicates that the new unit of measurement will contain the exponents of the dimensions (1 exponent for metres and -2 exponents for seconds, since the seconds are squared in the denominator).

This way of representing base and derived units allows for an infinite representation of units of measurements, despite the fact that some of them may be used to represent phenomena in the physical world. Overall, there are a total of 22 derived units recognised by the SI, some of them are shown below [10]:

<table>
<thead>
<tr>
<th>Name</th>
<th>Symbol</th>
<th>Quantity</th>
<th>Base Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>hertz</td>
<td>Hz</td>
<td>frequency</td>
<td>second⁻¹</td>
</tr>
<tr>
<td>newton</td>
<td>N</td>
<td>force, weight</td>
<td>kg.metre.second⁻²</td>
</tr>
<tr>
<td>pascal</td>
<td>Pa</td>
<td>pressure, stress</td>
<td>kg.metre⁻¹.second⁻²</td>
</tr>
<tr>
<td>joule</td>
<td>J</td>
<td>energy, work, heat</td>
<td>kg.metre².second⁻²</td>
</tr>
<tr>
<td>watt</td>
<td>W</td>
<td>power, radiant flux</td>
<td>kg.metre².second⁻³</td>
</tr>
<tr>
<td>square metre</td>
<td>m²</td>
<td>area</td>
<td>metre²</td>
</tr>
<tr>
<td>cubic metre</td>
<td>m³</td>
<td>volume</td>
<td>metre³</td>
</tr>
<tr>
<td>metre per second</td>
<td>m/s</td>
<td>speed, velocity</td>
<td>metre.second⁻¹</td>
</tr>
<tr>
<td>cubic metre per second</td>
<td>m³/s</td>
<td>volumetric flow</td>
<td>metre³.second⁻¹</td>
</tr>
</tbody>
</table>

In short, by combining several base quantities and raising them to certain powers, any derived quantity can be defined. These are called dimensional exponents [10]. The following syntax can be used to define dimensional exponents.

\[
\text{Derived} = \text{Baseof} \text{base} | \text{Timesof} (\text{derived} \ast \text{derived}) | \text{Expo}f (\text{derived} \ast \text{int})
\]
Dimensional exponents are another way to describe an already existing quantity in an abstract way. For example, the acceleration seen before can also be expressed as: \( \text{Times}(L, \text{Exp}(T, -2)) \). Which indicates that the base unit for length \((m)\) will be multiplied and raised by the negative squared exponent of the time unit \((s)\), which will produce \(ms^{-2}\) in concrete syntax, which is the derived unit for acceleration. Although units can be represented this way in software, this representation is not the only one, as the 7-tuple form may also be combined with other approaches to generate a more complete solution of representing units, such as in Kripser et al.’s work [7].

Alternatively, one may just code units in primitive types, which is the crux of a large problem and may lead to error prone code and is fundamentally in conflict with Object-Oriented programming and Domain Driven Design. Taking the speed of light in valid but different forms in C-type languages:

```java
static final double c = 1079252849;
static final double SPEED_OF_LIGHT = 1079252849;
static final double SPEED_OF_LIGHT_IN_KM_PER_H = 1079252849;
```

### 2.3 Unit Equivalence and Equality

It is important to make the following distinction of *equivalence* and *equality* when checking for unit correctness. Take \(1kg = 1kg\), that is to say, that 1 kilogram is *equals* to and is *equivalent* to 1 kilogram. And by definition, if both the left-hand side and the right-hand side have the same magnitude and unit, then the LHS and RHS are *equals*, which automatically entails equivalence. However, 1 kilogram is *equivalent* to but not *equals* to 1000 grams \((1kg \equiv 1000g)\), because while they might be equal in magnitude, the RHS does not have the same unit as the LHS, making them equivalent instead of equals. It would go without saying that \(1k\Omega\) would neither be equivalent nor equals to \(1000Nm\), as they are different units representing different natural phenomena.
2.4 Dimensional Analysis, Consistency and Unit Conversion Methods

When checking that physical quantities are added or equated to each other, the fundamental units and physical quantities used must be consistent so that an expression can be made between these terms. To verify that an equation is sound or correct from a perspective of units, a so called *dimensional analysis* is performed to check that the LHS and RHS end up simplifying to the same unit of measurement, ergo *commensurable* and dimensionally homogeneous. Dimensional analysis is important in the context of this thesis as many unit measurement libraries perform it whether directly or indirectly to check for unit correctness.

Manually performing checks to make sure units are compatible and that an equation is dimensionally correct is a tedious process which takes a long time. Therefore, there are a few methods and algorithms identified in literature which can be used to ensure that functions/subroutines, equations, units and variables are dimensionally correct or return the correct units. These methods differ from using inference to annotating code. Orchard et al. [12] implement an inference based method in FORTRAN using Gaussian elimination to constraints generated from source program code. Chen et al. [13] use rule-based analysis to ensure Dimensional safety and Fowler [3] and Kripser et al. [7] propose the Quantity and Physical Quantity pattern as a way to correctly and safely program units of measurements into systems to begin with. Hangal et al. [14] implement an automatic dimensional inference algorithm and formulate dimension inference as a polymorphic type inference problem where the dimensional relationships between variables in a program are inferred from the way they are used through intraprocedural inference rules. This is done by namely translating rudimentary program statements into their corresponding inference. As shown in figure 1 in the following page.
Figure 1: Dimensional inference of program statements

Since all units of the same dimension can be converted into one another, five possible unit conversion types (explicit or implicit) can be made depending on the statements made in a program and the possible reasons for their occurrence. If two statements were to be equated to each other using the same units but represented differently, then an implicit conversion can be made. Figure 2 on the next page shows the reasoning behind this.
Annotation based methods work by annotating units into the variables in code instead of inferring the units from the variables as seen from the implementation done in C#’s functional language implementation F#. Through annotations, dimensional safety can be ensured at compile-time. Although one disadvantage of annotation driven dimensional correctness is that it may impose a burden on developers to annotate their variables and functions/subroutine and method calls. Therefore, the method by inference is generally preferred as it less tedious than manually annotating programs.

<table>
<thead>
<tr>
<th>conversion type</th>
<th>example</th>
<th>possible reasons</th>
<th>im./ex.</th>
</tr>
</thead>
<tbody>
<tr>
<td>diff. dimensions, diff. exponents</td>
<td>(N) and (Nm)</td>
<td>formula wrong, operators mixed, operands forgotten</td>
<td>✗ ✗</td>
</tr>
<tr>
<td>same dimension, different units, c-factor ≠ 1</td>
<td>feet ((ft))</td>
<td>diff. software</td>
<td>✗ √</td>
</tr>
<tr>
<td></td>
<td>metres ((m))</td>
<td>components use different base units</td>
<td></td>
</tr>
<tr>
<td>same dimension, different units, c-factor = 1</td>
<td>(N)</td>
<td>same unit differently represented</td>
<td>✓ ✓</td>
</tr>
<tr>
<td></td>
<td>(kg \cdot m \cdot s^{-1})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>same dimension, same units, diff. prefixes (p_i), prefix products equal</td>
<td>kWh (= \frac{g \cdot km^2}{s^3} \cdot h)</td>
<td>same unit differently represented</td>
<td>✓ ✓</td>
</tr>
<tr>
<td>same dimension, same units, diff. prefixes (p_i), prefix products unequal</td>
<td>kWh (= \frac{kg \cdot km^2}{s^3} \cdot h)</td>
<td>diff. scales of same unit</td>
<td>✗ ✓</td>
</tr>
</tbody>
</table>

Figure 2: Conversion types and permissions
2.4.1 Language Support and Libraries

Dimension checking proposals generally do not exist in mainstream programming languages and generally, does not exist as a construct (despite the fact that they can, and should). The only support for dimensional checking in mainstream languages are Apple’s SWIFT language (NSMeasurement) [15] and F#'s support for unit of measurement [16]. Otherwise, UoM as an inbuilt feature is mostly lacking in the majority of programming languages, which is a cause for concern as it is quite a rudimentary feature for programming languages to have [17].

There does exists a plethora of libraries (collated in previous work [4]) that can provide developers with support for units, scan existing codebases using dimensional analysis to measure commensurability of units. There exists many ways to compute the dimensional commensurability of programs by using Gaussian Elimination [18] or using rule-based methods devised by Chen et al. [13] or using a model-driven approach in a created language (CellML) [19], or even using an object-oriented approach [20]. Other methods do not even require units in programs to be annotated and uses inductive inference [14] [21] compared to easier methods that do require units to be specifically annotated [5] [8] but sacrifices convenience for sake of being explicit. Frameworks that exist to prevent the problem by being proactive (e.g: by implementing an improved version of the Quantity Pattern, the Physical Quantity Pattern [7]) utilise proper knowledge and object oriented modelling.

2.4.2 Possible Reasons for General Lack of Adoption

One reason as to why this is the case is that most programmers might not have heard of the concept of dimensional safety and would not have given it a thought. Some might like the idea in principle but might not be bothered about it. It is also very likely that a lot of programmers have not realised that these checks could be made. Or that they would not like the extra checks and safety features to be made as they would be more computationally expensive due to some specific unit conversion method, or inefficiencies in the method of dimensional analysis. Lastly, it is also possible that a small minority use these methods and are happy with it, but are not in a social position to influence others to adopt these methods. All of these initial postulations will weigh into the design of the data collection methods in the following methodology section.
2.4.3 JSR-385 Programming

The theory chapter also includes a brief overview of JSR-385, one of the main UoM libraries on the internet in order to learn more about how UoM solutions work and to help in formulating the questions behind the data collection methods. It is also a well-rounded state of the art library which is a good representative of how UoM libraries work.

Before getting to the methodology section, some time was spent getting familiar with one of the main unit of measurement libraries (among thousands) that exist on the internet. The main reason for doing so is so that I could get a feel for how unit measurement libraries work and understand their advantages and disadvantages, this would aid in formulating questions for the surveys and questionnaires as it would provide a different perspective. Another reason is so that interviewees understand how concepts such as quantities, dimensions and units are represented in code (if they are not already familiar with de-facto standard unit measurement libraries). One library on the web that has an excellent coverage over these concepts is the JSR-385 [22] specification (which is itself, an upgraded standard from JSR-365). From the reference manual, JSR-385 describes itself as "A framework supporting robust representation and correct handling of quantities" which "is a basic need of Java developers across domains including science, engineering, medicine, finance or manufacturing". In the specification, it is acknowledged that developers either have to "use an inadequate model of measurement or to create a custom solution, which can lead to significant programmatic errors", these errors being unclearly coding units into variables (and having to infer the context behind the variables to figure out the type of units). The specification lastly concludes it proposes a "standard solution which is both safer and efficient and saves developer time for domain specific work".

From the specification, JSR 385 proposes to establish safe and useful methods for modeling physical quantities. This includes:

- Interfaces and abstract classes with methods supporting unit operations, including:
  - Checking of unit compatibility
  - Expression of measurement in various units
  - Arithmetic operations on units

- Concrete classes implementing standard unit types (such as base and derived) and unit conversions
Concrete classes for parsing and formatting textual unit representations

A “database” or equivalent repository/system of predefined units.

A concise explanation of how the API works is that it revolves around the `Unit` interface in the `javax.measure` package. A unit in this context represents main fundamental quantity types such as ’kilometres’ or ’watts’. Using Java generics, unit types can be parameterized with a dimension e.g: A unit of length or and a unit of time which can be coded as respectively:

```java
Unit<Length> m = METRE; Unit<Time> s = SECOND;
```

Using these units, it is possible to convert numbers and double from one unit type to another using the `UnitConverter` class type. Consider the following where units are converted from metres to centimetres (4 metres to 400 cm):

```java
Unit<Length> m = METRE;
Unit<Length> cm = CENTI(METRE);
UnitConverter mToCmConverter = m.convertTo(cm);
Number resultInCm = converter.convert(4);
```

The unit conversion performed above only works if the units have the same dimension (in this case, length). This means that it is not possible to convert a variable of type:

```java
Unit<Length>
```

into a variable of type:

```java
Unit<Time>
```

because they are of different dimensions. This sort of illegal type conversion is caught by the Java compiler, making writing code with a UoM library much safer and cleaner at compile-time. The features of Java generics also lend themselves to UoM libraries and makes development easier in Java, therefore, Java becomes a good natural choice of programming language for developing UoM libraries and consequently perform calculations and computations with units and unit conversions.
3 Methodology

In this chapter, the methodology used when conducting the thesis is outlined. The general methodological approach to describe the nature of the study and methodology, lastly concluding with describing the form of the data collection methods and how they will be carried out.

3.1 Methodological Approach

The general methodological approach taken in this study is akin to an explanatory study, or an exploratory research similar to research conducted earlier by Björkman et al. used to categorize UoM libraries [4]. Because the area has not really been properly defined yet with little research made on the lack of adoption of UoM libraries, this type of methodology is used to explore, generate data, and research by conducting interviews and gathering data from questionnaires in that said area. The outcome of such a study is to provide a foundation for more research later [23]. In addition to providing a foundation for more research later, the outcome of such a study falls under Design Science Research (DSR) as the intended outcome is to recommend ‘specific guidelines for evaluation and iteration within research projects’ so that a produced artefact can more effectively solve the problem [24]. Hevner presents the following 7 guidelines for DSR [24] for research in IS:

1. Design as an artefact
2. Problem relevance
3. Design evaluation
4. Research contributions
5. Research rigour
6. Design as a search process
7. Communication of research

The work from this thesis is enabled by these points as it would contribute to the design or betterment of existing designs (in this case, UoM libraries and related artefacts) as these
solutions become better developed to important business problems. Rigorous methods of data collection will be employed (which will be discussed in the next subsection) to evaluate the state of UoM libraries with an outcome producing knowledge for further software artefacts of this kind.

3.2 Methods of Data Collection

The aim of this section is to devise the appropriate data collection methods to collect both qualitative and quantitative data to answer the research questions of this thesis. It is a given that since the lack of adoption of UoM libraries has not been researched before, data will have to be generated entirely through primary data collection methods such as questionnaires and interviews. Therefore, the following data collection methods were chosen:

- Web questionnaires
- Personal/Online semi-structured interviews
- Discussions through forums and bulletin boards

The benefits of a questionnaire here are quite advantageous as they are both time and cost effective [25]. A web questionnaire makes it even easier to reach out to respondents in different geographical areas of the world, which is particularly useful in the limited time frame of this thesis.

The interview is also another relatively straightforward approach to gathering data in IS that is mainly qualitative in nature. It can be structured, semi-structured or unstructured and should ideally encourage open-ended responses. Interviews are highly interactive and carrying out interviews for this thesis is advantageous as it is a useful method for probing responses and gathering detailed information in a field where reasons for the lack of adoption of UoM libraries has not fully been explored or examined yet. One drawback of interviews is finding potential respondents, but by utilizing social networks and pinpoint respondents on the web for interviews, it will be possible to conduct interviews in the time frame of the thesis.

The criteria for finding potential interview respondents is that they must have been directly or indirectly involved with units of measurement and are ideally acquainted with UoM libraries, unit conversions or unit failures. The respondents should also be at least familiar with UoM
concepts such as SI base and derived units, dimensional consistency, and illegal unit conversion, etc... These criteria are established so that interview respondents give responses that are generally relevant to the topic and can be analysed and reasoned about in a systematic way.

3.2.1 Formulation of Questionnaire

The aim of the questionnaire is to collect both qualitative and quantitative data to answer the research questions of this thesis. A great deal of the data to be gathered from the questionnaire will be qualitative rather than quantitative as qualitative data analysis is more effective in this context where responses to the lack of adoption of these UoM libraries can be more deeply introspected. Also, since reasons for the lack of adoption are so varied, a mainly quantitative approach will not work as well as a mainly qualitative approach.

When devising the questionnaire, it is important to get a good sense of who will be taking the questionnaire as it is vital to capture all relevant qualitative and quantitative data from the right people.

3.2.1.1 Respondent Role: Managers, TTL’s and Developers

The organizational structure of a software development section of an IT company is traditionally formed of managers who manage product owners, who have multiple software developers (as well as QA testers) under their wing. This sort of structure may or may not be present in scientific communities, but the roles are generally well defined and a scientific computing developer might as well be considered a 'developer' in this sense. A manager in a scientific group may lead multiple technical team leaders who lead multiple scientific computing developers. Therefore, the questionnaire should capture the entire relevant population of interest if people with these roles are targeted by the questionnaire. Thus, it is important that the questionnaire starts off by questioning respondents what role they belong to so as to introspect further based on the role of the respondents. This is useful because a manager might have quantified the cost of unit failures and a technical team leader might have specific guidelines in dealing with unit errors in his/her team (which is something worthy to note) in the case that a UoM library has not been adopted. So more specific questions relating different aspects of the lack of adoption of unit measurement/conversion libraries may be asked depending on the role of the respondent.
3.2.1.2 Respondents: Adopters and Non-Adopters

Continuing on from the first question, the respondents can then be asked whether they have or have not adopted unit measurement libraries in their solutions. In the case that they are adopted, the respondents must explain why the lack of use is apparent. In the case the respondent has not adopted a UoM library, the respondent must explain the reason behind their choice. Additionally, respondents that fall under the adopter subset must note the benefits of adoption as well as give reasons as to why they did adopt a UoM library of their choice. Whilst respondents that are non-adopters must explain the reasons to why did not adopt a UoM library, in addition to answering questions related to the frequency of unit conversion errors, testing, and interference’s of unit conversions with coders practices. This is so that a need for UoM libraries could be identified among non-adopters, despite the fact that they have not actually adopted one. Therefore, the decision to not adopt a UoM library despite the need to do so could come under scrutiny when analysing the results.

3.2.1.3 Form and Flow of Questionnaire

With the above in mind, the questions in the questionnaire can be detailed and formalized in the following list:

- **Respondent Role:**
  - Manager (M)
  - Technical Team Leader (TTL)
  - Developer (D)

- **Manager:**
  - Do you use Unit Measurement Libraries within dept? (Yes)
    - **M-Y1:** Why has it not been adopted earlier?
    - **M-Y2:** Why do you think others would not adopt them?
    - **M-Y3:** Noted benefits of adoption?
    - **M-Y4:** Provide a quantitative figure on costs saved.
- Do you use Unit Measurement Libraries within dept? (No)
  * M-N1: Why has it not been adopted?
  * M-N2: Cost of errors (Ordinal)
  * M-N3: Interference with coders practices (Ordinal)
  * M-N4: Could you quantify the error in your local currency?
  * M-N5: Would you consider adopting a UoM library? (Ordinal)

• TTL:

- Do you use Unit Measurement Libraries within the team? (Yes)
  * TTL-Y1: Why has it not been adopted earlier?
  * TTL-Y2: Why do you think others would not adopt them?
  * TTL-Y3: Noted benefits of adoption?

- Do you use Unit Measurement Libraries within dept? (No)
  * TTL-N1: Why has it not been adopted?
  * TTL-N2: What guidelines are in place for to prevent unit conversion failures?
  * TTL-N3: How interfering are unit conversions and failures with coders practices? (Ordinal)
  * TTL-N4: What type of solution do/would you prefer?
  * TTL-N5: Would you consider adopting a UoM library? (Ordinal)

• Developer:

- Do you use Unit Measurement Libraries as a developer? (Yes)
  * D-Y1: Why has it not been adopted earlier?
  * D-Y2: Why do you think developers would not adopt them?
  * D-Y3: Noted benefits of adoption?
  * D-Y4: Would you recommend others to adopt UoM libraries? (Ordinal)
  * D-Y5: How influential are you to get others to adopt these libraries? (Ordinal)
  * D-Y6: To what degree is testing easier with a UoM library? (Ordinal)
– Do you use Unit Measurement Libraries within dept? (No)

* D-N1: Why has it not been adopted?

* D-N2: To what degree is testing difficult with regards to unit measurement errors and failures? (Ordinal)

* D-N3: How interfering are unit conversion errors and failures with your practices as a developer? (Ordinal)

* D-N4: What type of solution would you like to see? (Select between choices or provide more)

* D-N5: In hindsight, would you consider adopting a UoM library? (Ordinal)

* D-N6: How frequent are unit conversion errors and failures during testing? (Ordinal)

As seen above, the questions diverge based on the answer given (yes/no) to each respondent. This flow for the questionnaire is important as all qualitative viewpoints regarding adoption and/or lack of adoption must be captured respectively from both adopters and non-adopters of UoM libraries. Quantitative data is to be gathered from questions that are accompanied by 'ordinal' in brackets, otherwise the data gathered will be of qualitative form.

With the questionnaire fully formulated, it can now be created and hosted on the internet. One very popular online tool for quick prototyping and creation of questionnaires is Google Forms [26]. It is useful as it is intuitive to set up and allows for easy ways of organising and displaying data from collected responses.

### 3.2.2 Formulation of the Interview Template

As previously mentioned, the interview is a semi-structured interview. In an unstructured or semi-structured interview, there is an incomplete script. This means that questions may need to have been prepared beforehand, but generally there is a need for improvisation and a strict compliance to the interview questions is not necessarily required.

To begin with in every interview, the interviewee is introduced to the interviewers background, field of study, and purpose of the interview. This is kept short and brief so that it does not consume much of the interviewees time. Consent is taken so that their voices may be recorded and transcribed for analysis and coding.
Following shortly after, the interview questions depend on two different cases. **Case 1** (C1), where a UoM library has not been adopted and a custom solution was developed instead, and **Case 2** (C2), where a UoM library has been adopted. In C1, introspectiveness is required and the reason as to ‘why’ that is the case must be fully explored. Because this is a semi-structured interview, questions under C1 **may or may not** include the following (**and are not limited to**):

- **C1Q1**: How interfering are unit conversion errors in your experience (or to anyone around you) and with their coding practices? Did you have to sacrifice code readability for the sake of unit correctness (for example)?

- **C1Q2**: Why did you decide to use a custom, own solution?

- **C1Q3**: How effective are the guidelines in place in dealing with any unit conversion errors?

- **C1Q4**: How costly were unit conversion errors in the past?

- **C1Q5**: Lastly, do you recommend that these libraries be used in the wider scientific community?

In C2, the interviewee says that a library is used, so it is important to explore why it was not adopted earlier, note the benefits of adoption, and try to estimate a figure on the amount of money saved using unit measurement libraries. Again, since this is a semi-structured interview, questions under C2 **may or may not** include the following (**and are not limited to**):

- **C2Q1**: How beneficial was the adoption? How is testing easier?

- **C2Q2**: How much cost was saved by adopting unit libraries?

- **C2Q3**: Why do you think others might not use unit measurement libraries? Give reasons.

- **C2Q4**: Do you recommend others to adopt these libraries in the scientific community?

The goal for the interviews is to be around 20-30 minutes long, this is an ideal length of time to explore topics and themes to be latched onto in detail without generating any sort of fatigue on the interviewees behalf.
3.2.3 Forum and Bulletin Board Discussions

Another way to potentially gather considerable amounts of qualitative data is to encourage debate and discussion on online forums and bulletin boards such as Reddit [27] and Github [28]. This approach was chosen because it is easier to reach out to adopters and non-adopters of UoM libraries through the internet and consequently, listen to the rationale behind their choices and decisions. This was also an alternative for people who do not like taking questionnaires as some respondents might feel constricted by the format of a fixed questionnaire. Many respondents feel that questionnaires are too ‘constricting’ do not like taking one, and would rather describe the issue in a more free manner, thus gaining more potential qualitative data that would have otherwise not been expressed.
4 Results

In this section, the results of the study are divided into two sections corresponding to the data collection methods used from the questionnaires, and discussion boards. The results from the interviews are discussed in the next section. The results were gathered, cleaned, and consolidated in spreadsheet and statistical tools such as Google Sheets, Excel and IBM SPSS. ‘Cleaning’ and ‘consolidation’ involved removing nonsensical, empty, or answers that did not contribute to the study. Qualitative data that was gathered from the questionnaire from answers that demand written answers were isolated and assigned to data of qualitative category.

4.1 Web Questionnaire

During the timespan of the online questionnaire, 91 responses were garnered in total. Of the 91 respondents, 67% (N=61) of the respondents were developers, 25.3% (N=23) respondents were TTL’s, and lastly 7.7% (N=7) of respondents were managers. This is shown in figure 3.

What is your role within your team/department/company?

91 responses

![Figure 3: Role](image)

Manager/Business Unit Leader
Technical Team Leader or Group Leader
Developer
With regards to adoption or lack of adoption of UoM libraries. 49.2% of developers answered 'No' (N=30), and the remaining 59.8% of respondents answered 'Yes' (N=31), seen in figure 4.

**Do you use Unit Measurement Libraries as a developer?**

![Pie chart showing 49.2% Yes and 50.8% No](image)

Figure 4: Developer Adopters and Non adopters

47.8% of TTL’s answered 'No' (N=11), and 52.2% of TTL's answered 'Yes' (N=12), seen in figure 5.

**Do you use Unit Measurement Libraries within your team?**

![Pie chart showing 47.8% Yes and 52.2% No](image)

Figure 5: TTL Adopters and Non adopters

Lastly, in figure 6, 71.4% of Managers answered 'No' (N=5), and the 28.6% remaining answered 'Yes' (N=2).
In this section, the results from each of the roles (Manager, TTL and Developer) that have adopted UoM libraries are collated, which means that questions that are similar such as TTL-Y1, D-Y1 and M-Y1 are grouped together under one result for formatting and convenience’s sake. Therefore, the set of questions that will be collated (in addition to the non-collated questions) under the adopter-Y category are:

- Why was the reason to adopt Unit measurement libraries not considered earlier? (TTL-Y1, D-Y1 and M-Y1)
- Why do you think others would not adopt them? (TTL-Y2, D-Y2 and M-Y2)
- Noted benefits of adoption? (TTL-Y3, D-Y3 and M-Y3)

We begin with presenting the collated results first. For (TTL-Y1, D-Y1 and M-Y1), qualitative data was gathered since the question required a long answer text. The data was collated from most of the respondents who answered ’Yes’ (N=45). The data was cleaned and 37 useful qualitative data points were obtained from the 45 respondents, the data is available in the appendix under table 8.
Regarding (TTL-Y2, D-Y2 and M-2), a set of predefined options were available such as:

- Laziness
- Effort of learning and using unit measurement libraries
- Local solution already exists
- Lack of programming knowledge
- Performance specific inefficiency concerns (methods of computing Dimensional analysis/correctness and conversions)
- Not bothered about it
- Would rather implement own solution
- Current solutions are too clunky and complex/coarse grained
- Didn’t fit in the workflow
- Local solution already exists
- Solutions interfere too much with code
- Compile time performance
- "Provide text" (Qualitative)
The following results were obtained (where N = 45, the total number of respondents that answered ‘Yes’ to UoM adoption) and displayed into figure 7.

![Figure 7: (TTL-Y2, D-Y2, M-2) Why is there a lack of adoption?](image)

With regards to (TTL-Y3, D-Y3 and M-Y3), these set of predefined options were available in the form:

- Support for runtime handling and printing of units and quantities and dimensions
- Support for Compile time checking
- Faster development time
- Higher degree of interoperability
- Support for user-defined dimensions and units and quantities and systems of units
- Higher Performance
- Readability
- Support for strict symbol parsing including Metric prefixes and other cultural or specific prefixes with all units
- "Provide long answer text" (Qualitative)
The following results were obtained from N=45 respondents, the total number of respondents that answered 'Yes' to UoM adoption into figure 8

![Benefits of UoM adoption](image)

**Figure 8: (TTL-Y3,D-Y3,M-Y3) Benefits of UoM adoption**

Moving on from the collated results, the only non-collated results that remain are from (M-Y4, D-Y4, D-Y5, D-Y6). Unfortunately, M-Y4 garnered no data whatsoever on costs saved from using UoM libraries, so it is to be discarded. (D-Y4, D-Y5, D-Y6) are from the developers who answered 'Yes' to UoM adoption (N=31), where the questions are intended to respectively measure recommending UoM libraries to others, social influence to get others to adopt UoM libraries, and lastly the degree to which testing is easier with a UoM library.
Shown below is figure 9 on recommending UoM libraries, where 38.7% (N=12) of respondents would recommend, 22.6% (N=7) undecided, 6.5% (N=2) would recommend against, and 32.3% (N=10) would strongly recommend others to use and adopt UoM libraries.

![Figure 9: Would you recommend others to adopt and use unit measurement libraries?](image_url)

The next figure, figure 10 shows how influential developers are to get others to adopt UoM libraries so that maybe the lack of adoption of UoM libraries can be attributed to not enough social impetus. 42.5% of respondents (N=14) responded that they were somewhat influential while 9.7% (N=3) responded with very influential and 3.2% (N=1) respondent replied with extremely influential. 16.1% (N=5) of respondents are slightly influential. With 25.8% (N=8) of respondents were not at all influential.

Of the last adopter-Y questions, 41.9% of developers (N=13) say that testing becomes 'easy' with a UoM library. On the other hand however, 41.9% of developers answered 'neutral', meaning it makes no difference when testing unit based code, with 3.2% answering it was 'very difficult' even. Lastly, 12.9% of respondents (N=4) 'very easy', as shown in figure 11.
Figure 10: **D-Y5** How socially influential are you?

Figure 11: **D-Y6** To what degree is testing easier?
### 4.1.2 Role-Nonadopter (N) Results

In this section, the results from each of the roles (Manager, TTL and Developer) that have not adopted UoM libraries (M-N, TTL-N, D-N) are collated in a similar fashion to the previous section. The results from the questions that are collated together are those with a similar number:

- Why hasn’t a UoM library been adopted? (TTL-N1, D-N1 and M-N1)

- How interfering are unit conversions and failures with coders practices? (Ordinal) (TTL-N3, D-N3 and M-N3)

- What type of solution do/would you prefer? (TTL-N4, D-N4)

- In hindsight, would you consider adopting a UoM library? (Ordinal) (M-N5, TTL-N5, D-N5)

Once again, we begin with presenting the collated data first starting with (TTL-N1, D-N1, M-N1). For (TTL-N1, D-N1 and M-N1), qualitative data was gathered since the question required a long answer text similar to the Y1 questions. Respondents were also able to tick predefined checkboxes related to the disuse of UoM libraries, these options were:

- Laziness

- Effort of learning and using unit measurement libraries

- Local solution already exists

- Lack of programming knowledge

- Performance specific inefficiency concerns (methods of computing Dimensional analysis/correctness and conversions)

- Not bothered about it

- Would rather implement own solution

- Current solutions are too clunky and complex/coarse grained

- Didn’t fit in the workflow
- Local solution already exists
- Solutions interfere too much with code
- "Provide long answer text" (Qualitative)

The data was collated from most of the respondents who answered 'No' (total of N=46), and the bar chart figure 12 was produced.

![Bar Chart]

Figure 12: **(TTL-N1,D-N1,M-N1)** Why hasn’t a UoM library been adopted?

Moreover, qualitative data was obtained from the last checkbox alternative, which produced a significant amount of qualitative data. The data was cleaned and 57 useful qualitative data points were obtained from the 46 respondents, this data can be seen in the appendix under table 9.
The next collated question \((M-N3,TTL-N3,D-N3)\) deals with the interference of unit errors and failures with coders practices on an ordinal scale. From the 46 nonadopter respondents, bar chart and figure 13 was produced.

![Figure 13: \((M-N3,TTL-N3,D-N3)\) How interfering are unit errors?](image)

Questions \((TTL-N4,D-N4)\) are collated together from only TTL's and Developers, which deals with the preferred solution that they would like to see in which they are able to pick from two choices (in addition to providing more choices of their own qualitatively, where 9 points were obtained), this can be seen in figure 14 and table 3.

![Figure 14: \((TTL-N4,D-N4)\) Preferred solution?](image)
Table 3: **TTL-N4,D-N4** Extra answers

**Preferred Solution?**

A solution that is statically enforced through the type system where all operations are disallowed except those that you actually need.

Light type annotations that can be disabled for zero compiler overhead during local iteration; allow full checks in CI builds

Light weight

The one the client pays for

Not sure what either of these would look like. I like the java.time library, and would like something similar for other units.

Depends on problem

ints and floats

I think you should use a consistent unit system.

Static compile-time checks

Lastly, results from (M-N5,TTL-N5,D-N5) are collated together which asks respondents (N=46) whether they would now consider adopting a unit measurement library in hindsight after answering their respective ‘N’ questions. Figure 15 was obtained.

![Bar chart showing responses](image)

**Figure 15: (M-N5,TTL-N5,D-N5)** Consider adopting UoM libraries in hindsight?
Continuing on from the collated questions, the remaining D-N questions to be looked at are **D-N2** and **D-N6** which respectively deal with the difficulty of testing with regards to unit measurements and errors, and the frequency of unit conversion errors and failures during testing on an ordinal scale. The following barcharts were produced from the developers who answered 'No' (N=30) into figure 16.

![How frequent are unit conversion errors and failures during testing?](image)

Figure 16: **DN-6** Frequency of unit conversions and errors during testing?

To clarify figure 16. 20% of respondents (N=6) answered 'Never' while 50% of respondents (N=15) answered with 'Rarely frequent'. 26.7% (N=8) of respondents though that unit errors were 'Sometimes frequent', 3.3% (N=1) answered 'Often frequent' with no one answering 'Always frequent'.
In figure 17. 13.3% of respondents (N=4) answered that testing was 'Very easy' with regards to unit measurement errors and failures, whilst 16.7% (N=5) of respondents answered 'Easy'. 53.3% (N=16) of respondents were 'Neutral' on the matter and 16.7% (N=5) of respondents answered 'Difficult', with no respondent answering 'Very difficult'.

Figure 17: DN-2 To what degree is testing difficult?
Regarding TTL-N questions, only TTL-N2 would remain which is a question that demands a long answer text on guidelines in place that TTL’s use to prevent unit conversion failures. 10 qualitative data points were obtained from this question, tabulated into table 4:

<table>
<thead>
<tr>
<th>Guidelines</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>The conventions used in the software framework are clear, consistent, and long established.</td>
<td></td>
</tr>
<tr>
<td>We made our own library.</td>
<td></td>
</tr>
<tr>
<td>Don’t convert. Use a consistent system.</td>
<td></td>
</tr>
<tr>
<td>Use SI units except for display</td>
<td></td>
</tr>
<tr>
<td>Only very simple units used. Conversion errors break unit / integration tests.</td>
<td></td>
</tr>
<tr>
<td>Only use one set of units (meters for length, seconds for time, etc.), clear naming, unit tests around any math-heavy code</td>
<td></td>
</tr>
<tr>
<td>Units are tagged at module interface level, tags are checked at runtime. In our domain, units mismatch are a common issue (dealing with imperial and SI), we always keep units in mind.</td>
<td></td>
</tr>
<tr>
<td>Thorough unit testing against analytically solvable test cases.</td>
<td></td>
</tr>
<tr>
<td>We label output data with units religiously. Internal variables are named with units included when necessary</td>
<td></td>
</tr>
<tr>
<td>Give only one type of unit as input: $M^3$, $M^3/h$, $MW$, $h$</td>
<td></td>
</tr>
</tbody>
</table>
Lastly, only two M-N questions are left from the managers who answered 'No' (N=5). **M-N2** (determining the cost of errors on a ordinal scale) and **M-N4** (a question that demands a short answer text on a quantifiable figure on the cost of unit errors, failures and conversion). For **M-N2**, 2 respondents answered that it was 'Not at all' with 2 answering 'Of little cost'. 1 respondent answered that it was 'Somewhat Costly'. No respondent answered with 'Costly' or 'Very Costly'. Figure 18 was produced for **M-N2**:

![Figure 18: MN-2: Cost of unit conversion errors and failures?](image)

For **M-N4**, unfortunately not enough data was produced to highlight the costs of unit errors and failures.
4.2 Interview Material

A total of 3 interviews were recorded and were then transcribed into three documents (corresponding to 3 different interviews with 4 experts) and then stored locally. The interview material is ready to be coded and analysed for the next section of this thesis in the discussion and analysis section. The material of the interview documents is strictly kept confidential to preserve the anonymity of the interview subjects. If a request for the interview material is arises, then the supervisor of this thesis is to be contacted.

4.3 Forum and Bulletin boards

As mentioned in the methodology section, the forum and bulletin boards was an option for those who were not keen on taking the questionnaire. This option was available by default because the questionnaire was spread on forum and bulletin boards such as Reddit and Github, where forum goers can freely express their thoughts and ideas on the subject. The discussions on the forums and bulletin boards generated a great deal of data, which saved a considerable amount of time in the timespan of this thesis as it did not require any transcribing. The discussion comments were collated into a single document and stored so that they could be coded and analysed in the next section.
5 Discussion and Analysis

5.1 Coding

Using Saldana’s work as a reference [29] and using qualitative data analysis software such as MAXQDA2018, an elaborate coding scheme was created from the qualitative data i.e: data from the interviews, long answer texts from the web questionnaire and the discussion comments from the forums. Since little is known in this subject area and an exploratory approach was decided on as a methodology, inductive coding was used to create the codebook from scratch from the qualitative data. This entails dividing the qualitative data into smaller units, assigning codes to those smaller units of data based on its content, assigning those codes to overarching categories or families. Categories are in themselves codes that consist of a family of related codes. As a rule of thumb, patterns in the coded data are searched for in order to categorise them, these patterns according to Saldana [29] are characterised by:

- similarity (things happen the same way)
- difference (they happen in predictably different ways)
- frequency (they happen often or seldom)
- sequence (they happen in a certain order)
- correspondence (they happen in relation to other activities or events)
- causation (one appears to cause another)

Since coding is a cyclical craft, coding of the data was done in multiple passes. In the first pass, the qualitative data in the documents is primitively coded based on its content and patterns begin to emerge. In the second pass, the codes are grouped together based on the emerged patterns and categories are created. Further passes may be required to refine the categorisations of the codes and an effort is made in order to ensure that codes belong to their proper, respective categories. Further passes may also be required to appropriately rename the codes to better reflect the data it represents.

After many code passes, following code categories and their subcodes were induced (where code categories are boldened while codes are not). These table of codes can be seen in the appendix (tables 10 to 14).
To clarify a major difference between table of codes under LACK OF ADOPTION and table of codes under UOM PROBLEMS, the LACK OF ADOPTION table has more to do with the social issues surrounding the lack of adoption of UoM libraries, whilst UOM PROBLEMS are mostly technological problems that are inherently wrong with UoM libraries.

5.2 Discussion of Results and Analysis

Taking the quantitative data from the web questionnaire and the qualitative data from it, the interviews and the forums in mind, there is quite a lot of ground to cover here. To make it simpler, the analysed data from each of the data collection methods are discussed individually in their own section. To begin with, the data from the questionnaire (qualitative and quantitative), interview (qualitative) and bulletin boards (qualitative) are discussed in that specific order.
5.2.1 Discussion of Questionnaire (Qualitative Data)

For the qualitative data that was received from the questionnaire, the long and short answer texts were collated each into their own documents depending on role, which means there were a total of 3 documents (TTL qualitative, manager qualitative, and developer qualitative). In total, 150 qualitative data points of varying sizes were received (most of which are short answer texts). As with the previous section, the most frequent codes that appear in the documents are outlined and the most important qualitative data points are described on a case by case basis. The following table shows the most frequent codes, their categories and their frequency of occurrence:

<table>
<thead>
<tr>
<th>Code Category</th>
<th>Code</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>LACK OF ADOPTION</td>
<td>No immediate need</td>
<td>15</td>
</tr>
<tr>
<td>LACK OF ADOPTION</td>
<td>Unaware</td>
<td>15</td>
</tr>
<tr>
<td>LACK OF ADOPTION</td>
<td>Have/Create their own solution</td>
<td>7</td>
</tr>
<tr>
<td>LACK OF ADOPTION</td>
<td>No gain</td>
<td>6</td>
</tr>
<tr>
<td>LACK OF ADOPTION</td>
<td>Simple use-case</td>
<td>5</td>
</tr>
<tr>
<td>COST/COMPUTATIONAL/COMPILE TIME</td>
<td>Time to compile</td>
<td>5</td>
</tr>
<tr>
<td>LACK OF ADOPTION</td>
<td>Lack of experience</td>
<td>5</td>
</tr>
<tr>
<td>LACK OF ADOPTION</td>
<td>Overhead to getting started</td>
<td>4</td>
</tr>
<tr>
<td>UOM PROBLEMS</td>
<td>Not good enough</td>
<td>4</td>
</tr>
<tr>
<td>CODING UNITS/GUIDELINES</td>
<td>Simple</td>
<td>4</td>
</tr>
<tr>
<td>LACK OF ADOPTION</td>
<td>Disagreements over standards</td>
<td>3</td>
</tr>
<tr>
<td>UOM DESIRED FEATURES</td>
<td>Lightweight</td>
<td>3</td>
</tr>
<tr>
<td>LACK OF ADOPTION</td>
<td>Cost of dependency</td>
<td>3</td>
</tr>
<tr>
<td>UOM PROBLEMS/LANGUAGE SPECIFIC</td>
<td>Type restrictions</td>
<td>5</td>
</tr>
<tr>
<td>LACK OF ADOPTION</td>
<td>Big effort with large codebase</td>
<td>2</td>
</tr>
</tbody>
</table>
It is interesting to see how the most common codes from the questionnaire table, table 5 are different with respect to the forum data table (table 7). The pattern here is that people who talk about the issue on forums and bulletin boards are probably more engaged with UoM and would directly critique it and talk about the state of unit libraries, while for the questionnaire respondents, it seems to be that they are mostly unaware of UoM libraries or have no need to use such libraries. It seems here that the forum goers tend to be more engaged in the issue and openly discuss as they are more outspoken about the issue in comparison to the questionnaire takers.

The following codes: 'no immediate need', 'unaware', 'have/create their own solution', 'no gain' and 'simple use-case' are quite pervasive throughout the documents as they are in the questionnaire. It seems many have not even considered the possibility of UoM libraries existing, it seems as if many respondents had not even heard or thought of it until the day the questionnaire was spread. The lack of knowledge of unit failures and errors and that there exists libraries to remedy these issues is a common, reoccurring theme throughout this thesis, which is reflected by the many who took the questionnaire. To emphasise this, one respondent mentioned that the issue is largely due to "ignorance of UoM solutions; i.e. where and when they should be used. As put by one respondents comment in the context of this thesis, this is not "[a] lack of programming knowledge" but rather "[a] lack of knowledge that a problem and solutions even exists". So it is apparent that many are not aware of the pitfalls of unit errors in numerical code. Additional answers ranged from it not being a problem when using a consistent set of units, too little of a concern for more "general codebases" where many things are without units, manual conversion functions "covering most cases", no-effort “sweet spot” solution not easily found, to none of unit related issues being addressed by adoption and were all a tax on subsequent use. For one respondent, it was not even seen as a crucial feature, but more of a 'nice-to-have'. This emphasises that people view UoM libraries differently and/or have differing use cases when using these libraries.

Computational costs were highlighted as a concern for some, the time to compile would drastically increase as compilation overhead would be "worth the benefit of just unit-testing math code" as "almost every numeric instance variable would have its own unique unit" as "template instantiation baggage is too costly for benefit". One concern was that even if a linear algebra support a UoM library, compile-times could "grow non-linearly with the number of dimensions in the problem to be solved, maybe $O(n^2)$ or $O(n^3)$", since the compiler would have
to derive a UoM type for every temporary variable, raising the rhetorical question of how many temporary variables with unique types there are in a matrix-multiply. The respondent goes on to say that "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". This is a legitimate concern as compile times for very large projects can be considerable, but the addition of even more "boilerplate code" could further compound compile times.

On the topic of types, type restrictions were a common concern of questionnaire respondents, for example, Boost.units ‘quantity<T>‘ (and others) do not preserve standard-layout, triviality and POD-ness (plain old data) of ‘T’, which are critical properties for the domain of one respondent. Matrices often require heterogeneous sets of measures and one respondent answered that "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". These worries highlight that unit types are not so standard yet and that there is a concern regarding the compatibility of these unit types by other libraries.

With relation to not 'being good enough' or 'overhead to getting starting' and 'lack of experience', there are quite a few tagged comments for these codes. Some comments range from UoM libraries causing "more pain than they relieve", "lack of understanding in teams", "not familiar with it" as they are "scared of compiler errors" and that there is "difficulty choosing among libraries and assessing whether a particular library fits the needs". Lastly, a respondent answered that a thorough code review would still be needed to detect mistakes. For example, a developer could write kilometersPerSecond instead of metersPerSecond and the error would not be caught automatically, so an argument is that including units in variable names achieves almost the same result. This is further compounded by UoM libraries being 'not idiomatic enough', according to one respondent: "many UoM libraries are non-idiomatic to be used (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 e.g. 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". So from this and the last paragraph, it is quite apparent that it is hard to integrate UoM solutions with respect to types
with other libraries, indicating a possible need for UoM solutions to implement commonly used interfaces, thus the need for a standard.

For the last few codes, 'big effort with large codebase’ and 'DB interactions’. The perceived effort cost of porting existing, 'working’ code to work with UoM libraries is quite considerable, one respondent noted that "the libraries always fail to interact with the rest of the ecosystem". Database interactions with UoM would be questionable as units would have to be stored in addition to the data. This potentially increases the total amount of data to be stored in databases which can be problematic for some developers. Lastly regarding runtime concerns, runtime handling of units were cited as less flexible than compile handling of units, as there is "no support for units that are modifiable at runtime as opposed to compile time". This is a good point, since many UoM libraries focus their concerns on compile time and not runtime handling of units.

All in all, convincing and compelling reasons were given to the lack of adoption of unit libraries from the qualitative data from the questionnaire. The main takeaways here are:

- unawareness and lack of concern are regarding UoM solutions were unfortunately very pervasive in the questionnaire, it seems as if questionnaire respondents were handling units their own way but were not concerned or were unaware that UoM solutions existed which could handle these issues to some extent

- concern over computational performance during compile time with regards to units and integration and performance of UoM libraries with maths and linear algebra libraries and that runtime handling of units are not good enough for some developers

- the overhead to getting started and the cost of dependency of a UoM library, in addition to type constrictions and lack of support for UoM libraries by other libraries

- the lack of need to adopt a library for most developers as they have guidelines such as using the same units, use few units or generally do not work in a field where so many units are used.
5.2.2 Discussion of Questionnaire (Quantitative Data)

The most interesting quantitative result is how alike the results from TTL-Y1, D-Y1, M-Y1 (why was the reason to adopt unit measurement libraries not considered earlier) and TTL-N1, D-N1, M-N1 (reasons to not use UoM libraries). It is almost as if a pattern appears. To clearly see this pattern, the predefined reasons are tabulated below (recall that total amount of adopters $N = 45$, nonadopters $N = 46$) with their differing frequencies:

<table>
<thead>
<tr>
<th>Reason</th>
<th>(Adopter-Y1)Frequency</th>
<th>(Nonadopter-N1)Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laziness</td>
<td>18</td>
<td>16</td>
</tr>
<tr>
<td>Effort of learning</td>
<td>19</td>
<td>15</td>
</tr>
<tr>
<td>Local solution already exists</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>Performance specific inefficiency concerns</td>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td>Not bothered about it</td>
<td>20</td>
<td>15</td>
</tr>
<tr>
<td>Would rather implement own solution</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Current solutions are too clunky</td>
<td>12</td>
<td>6</td>
</tr>
<tr>
<td>Didnt fit in the workflow</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>Local solution already exists</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>Solutions interfere too much with code</td>
<td>21</td>
<td>17</td>
</tr>
</tbody>
</table>

The most commonly frequent reasons such as: laziness, effort of learning, not being bothered about it, UoM’s interfering too much with code were already top, reoccurring concerns that were already highlighted in previous sections. It is of no surprise here to see how they dominate concerns regarding UoM libraries. This further highlights the fact how significant these reasons are and how much impact they have for people to adopt a unit library in their solutions.
Regarding how much easier testing is with a unit library for developer adopters (D-Y5), it is interesting to note that 41.9% thought that it was ‘easy’ and 12.9% ‘very easy’, with 41.9% of developers being ‘neutral’ on the topic. This signifies that UoM libraries can be quite impactful and aid testing or it might not at all (likely depending on what developers work with). It might be possible that 41.9% of developers do not work too much with units (even though they have adopted a UoM library) and would therefore be less willing to give an input regarding this matter.

In terms of benefits of adoption (TTL-Y3,D-Y3,M-Y3), most of the respondents chose ‘support for compile time checking’ with 30 responses and 23 for ‘support for runtime handling of and printing of units and quantities and dimensions’ and 20 for ‘Support for user-defined dimensions and units and quantities and systems of units’. Therefore, it is clear here that users not only need UoM libraries that support SI units and their derivatives, but also custom, user-defined dimensions and units.

Figuring out how interfering unit failures and errors is generally not a trivial task, (M-N3, TTL-N3, D-N3) tries to figure this out by asking the nonadopters how interfering they are. The data is essentially split in two halves where a total of 22 nonadopter respondents thought that they were somewhat interfering, interfering and very interfering, meanwhile the other 24 nonadopter respondents thought that they were of little interference to not being interfering at all. This again could vary depending on the field of work of the respondents, but it is significant considering that the questionnaire was spread to scientific communities and online boards.

Quite a few adopter developers recommended that others adopt a unit library in their solutions, with a total of around 71% of developers ‘recommending’ to ‘strongly recommending’ and 22.6% ‘undecided’. This indicates that unit libraries is something to generally recommend if one were to work with units. In fact, unit libraries should be part of a general API or language standard, like the C++ Standard library or Java API standard platform edition.

In addition to providing extra reasons as a long answer text, the ‘what type of solution would you prefer’ collated question (TTL-N4,D-N4) allowed respondents to pick from two predefined solutions. A ‘heavyweight’ solution, which is just like any UoM library where variables are given explicit units or are dimensionless and a compile and runtime checker ensures dimensional correctness, or a ‘lightweight’ solution, which is something that would
interact with the output of a program that has units and data and would do conversions (if needed) as a sort of middleware [17]. 17 respondents would prefer the heavyweight solution while 10 respondents would prefer the lightweight/middleware solution. This is an interesting result, but in hindsight, it is possible that the respondents did not understand what the 'lightweight' solution entailed or implied. It could also be that the question was poorly formulated in hindsight and the definition of 'middleware' was not fully communicated (as it was when discussing the concept of a middleware with regards to a unit library with the first interview subject).

Lastly, it is interesting to note the result for the collated questions (M-N5,TTL-N5,D-N5) where 7 respondents would now consider adopting unit libraries in hindsight, 10 would not and 29 might or might not consider doing so. It is possible that people who would not consider it do not really see themselves working with units all that much as nonadopters. For the hesitant ones, they might or might not consider using a UoM library depending on the circumstances, as there are quite a few variables to consider.

From this section, it is therefore apparent that:

- unawareness and lack of concern are also pervasive as it was in the previous section, in addition to that, it seems as if people are underestimating the severity of the issue and do not really see it as much of a concern, despite UoM solutions being recommended by most who use it (signifying that it is somewhat effective)

- the effort of learning these libraries may not provide a return on the investment of learning them for some

- in addition to compile time and runtime handling of units, UoM libraries should also provide some degree of support for custom unit types and user defined dimensions and units.
5.2.3 Discussion of Interview Data

By reaching out to scientists and developers both on the web and locally, a total of 3 interviews were conducted where 2 of the interviews were conducted on the web and 1 of them by interviewing a local scientist from Uppsala university. For the sake of anonymity and confidentiality, we shall instead refer to the interviewees by their subject number, the order number by which they were interviewed:

- **Interview subject number 1**: by starting with a lead with the help of the supervisor of this thesis, it was possible to reach out to a local scientist at Uppsala university who specialises in materials theory and scientific computing. A lot of ground was covered in this interview such as reasons to why the lack of adoption of UoM libraries is prevalent in scientific communities, costs of lack of adoption and frequency of errors related to units due to said lack of adoption.

- **Interview subject number 2**: the next interview subject was found by reading a Quora question relevant to the topic on UoM libraries followed by reaching out to the interview subject through email. The interview subject has a background in functional programming, programming language, and compiler design. In this interview, the lack of adoption of UoM libraries is discussed through the lens of programming language design and also from a software engineering perspectives. I.e: the ideal golden standard that UoM libraries should reach for was discussed and how a programming language should be designed to ease the barrier for entry and functionality of a UoM framework to be built on top of it. The lack of adoption of UoM libraries from a social and lack of technological push was also discussed in this interview.

- **Interview subject number 3 + 4**: this interview was also conducted on the web by emailing lead specification members and developers of a certain unit measurement standard for a very popular programming language. The idea here was to brainstorm reasons for why developers would not adopt their UoM framework, so the answers that were received in this interview were mainly constricted to that programming language but there is enough material to generalise the answers to more languages and frameworks. Additionally, successful adoption of this UoM library was discussed in multiple projects.
We begin with discussing the results from interview subject number 1, the local material scientist. In his field of research, a wide array of units are thought out and used depending on the scale of the physics looked at in order to tackle a new problem or explain phenomena. One step in tackling new problems in physics is by creating models using said 'thought out' units. However, the crux of the problem arises when trying to **compare the results of these models with other experimental groups** and relaying the information to other scientific groups who might have different intuitions for different unit sets. According to the interview subject, different scientific groups have different sets of units and different intuitions because certain kinds of problems can be simplified into certain equations by deriving them into subsets of SI units. Scientists then work within those equations and develop themselves within those subset of units into a 'community’. This leads to a divergence which is estimated to have started 50 years ago where scientific communities "started out with different parts and then started into different problems after 50 years of the same field". So essentially, different scientific departments have evolved their own conventions, methodologies and ontologies.

This is the interview where the majority of the codes under 'Different communities' in code table express themselves. However, this interview does not only tackle the issue of diverging scientific communities and their unit sets, but hints to gives legitimate reasons to the lack of adoption of UoM libraries. Codes under table (**LACK OF ADOPTION**) such as "Big effort with large codebase" and "Not being paid to develop code" are each expressed twice. Essentially, these codes represent in this text that scientists are not being funded to develop clean, elegant code that is free of unit errors and failures, but that they are being paid in research grants to make progress in a specific field. Therefore, writing or rewriting existing code to be free of unit failures and errors is a major undertaking as it could come at the expense of furthering progress in scientific research. According to the interview subject, even having to rewrite 10% of a large scientific codebase for the sake of units is a huge undertaking as mistakes can go in, testing of scientific software is also time consuming and many errors can go in.

Lastly, in this interview there, there are many codes expressed from table of codes where the interview subject talks about desired features that UoM libraries need to have to gain more traction. Such as being able to handle enormous amounts of dimensions without slowing down scientific models, have them be intuitive and easy to use and lastly, the creation for a sort of 'middleware', this middleware would entail wrapping software (e.g: the scientific computing), the wrapper is then accessed through an interface which can be used at the input segment to
control the data and the type of units that go in, and an output segment of the computing software so that developers can convert the output data from one unit type to another when desired. It is this hope for a middleware which would ease problems when scientists collaborate over different units. Interestingly, this follows Damevski’s reasoning where developers should not be burdened by units at each statements of their programs, but that units should instead be present in software component interfaces [9] [17].

In the second interview, the current state of UoM libraries were discussed, not only from the viewpoints of different intuitions from different use cases and communities, but also from a programming language, design and engineering perspective. Interview subject number 2 criticises the state of lack of adoption of UoM libraries from a technological and social standpoint.

From a technological standpoint, most UoM libraries are not intuitive to use by having bad usability and bad error messages. The interview subject gives an example of a very good Haskell UoM library that functions intuitively but produces ’nonsense’ for error messages. In addition to that, the interview subject argues that although UoM solutions exist for popular and mainstream languages such as JavaScript or Python, they are not very well suited for UoM solutions because they are dynamically typed languages, which means that the types of variables are not known at compile time but at runtime, which makes having to work with units in dynamically typed languages trickier. Additionally, there is a lot of overhead to getting started with a UoM solution as the cost of an additional dependency can outweigh the advantages of adopting a UoM library, especially if trying to adopt a UoM library with large existing codebases.

From a social standpoint, the interview subject argues in a straightforward manner that most people do not feel the immediate need to have something like this or are simply unaware of the existence of UoM libraries. Unawareness is a major part that leads to a lack of social push for adoption of UoM libraries. Even if scientists work with units in their code, he argues that it is very unlikely that they have heard of ’UoM libraries’ specifically. According to the interview subject who works in data science, the issue is not very important and the priority is not so high unless the field of work is in physics, mathematics or scientific code that tends to be very unit heavy with many dimensions. So unless rockets are being blasted into space, then there is no ’immediate need’ to adopt a UoM library for the majority of developers.
In the third interview, 2 lead specification members for a popular specification request were interviewed for reasons behind the lack of adoption of UoM libraries. In this interview, it seemed as if the interview subjects were less critical about the state of UoM libraries and more apprehensive to give answers regarding lack of use.

They identified positive cases where UoM libraries have been adopted (table of codes relating to successful implementation projects), but after some discussion, they were more open to brainstorming. Firstly, that there could be truth to the fact that the verbosity of the underlying language ("a lot of boilerplate code") could contribute to the difficulty of use of the UoM library, the developers stated that this is a concern and that a wrapper extension to the UoM library and its underlying language was built in Kotlin which offloads a few additional features that the base language doesn’t achieve. Additionally, runtime handling of units still remains a challenge for this UoM library, which is a requirement for many who use UoM libraries. Secondly, all too compelling reasons are given such as academics and developers not caring and/or disagreeing over standards and having to create a custom solution which one of the interview subjects even ended up doing in a C++ personal project of his.
To summarise with, the main takeaways from these interviews are the following:

- different scientific communities have different unit and different intuitions for those unit sets due to drift over time, tradition rooted in scientific communities could lead to a lack of adoption of UoM libraries due to old solutions being ’good enough’

- existing UoM libraries are rife with issues, leading to the following requirements to emerge: 1) be intuitive and easy to use without having a lot of baggage or having boilerplate code, 2) the ability to handle enormous dimensions at compile time (without slowing the speed of compilation of code), 3) the ability to easily integrate UoM libraries with existing code without having to rewrite the majority of the code and, 4) the ability to better handle units at runtime

- unawareness and the lack of immediate need to adopt a UoM solution

- despite potential benefits of UoM, there is an overhead to getting started with a UoM solution and the cost of having a dependency to a UoM solution may outweigh its benefits

- there is no push to adopt a UoM library because there is no ’impetus’, that is to say that people who are influential and are connected in the right way (according to interview subject number 2) have not pushed for or evangelised unit libraries as much as others have for other technologies (such as the Nix package manager, in his example).
5.2.4 Discussion of Forum Data

With around 67 qualitative data points of varying size in the collated document, there is a lot of information to go through. We begin by listing the most frequent codes that appear in this document and describe them on a case by case basis. The following table shows the most frequent codes, their categories and their frequency of occurrence:

<table>
<thead>
<tr>
<th>Code Category</th>
<th>Code</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>UOM PROBLEMS</td>
<td>Bad usability</td>
<td>10</td>
</tr>
<tr>
<td>LACK OF ADOPTION</td>
<td>Simple use case</td>
<td>9</td>
</tr>
<tr>
<td>UOM PROBLEMS/LANGUAGE SPECIFIC</td>
<td>Precision loss</td>
<td>6</td>
</tr>
<tr>
<td>UOM DESIRED FEATURES</td>
<td>Type flexible</td>
<td>6</td>
</tr>
<tr>
<td>LACK OF ADOPTION</td>
<td>Have/Create their own solution</td>
<td>6</td>
</tr>
<tr>
<td>LACK OF ADOPTION</td>
<td>Unaware</td>
<td>5</td>
</tr>
<tr>
<td>CODING UNITS</td>
<td>Clearer, compile-time safe code</td>
<td>5</td>
</tr>
<tr>
<td>LACK OF ADOPTION</td>
<td>Priority towards performance</td>
<td>5</td>
</tr>
<tr>
<td>UOM PROBLEMS/LANGUAGE SPECIFIC</td>
<td>Type restrictions</td>
<td>5</td>
</tr>
<tr>
<td>LACK OF ADOPTION</td>
<td>No gain</td>
<td>5</td>
</tr>
<tr>
<td>LACK OF ADOPTION</td>
<td>No immediate need</td>
<td>4</td>
</tr>
<tr>
<td>UOM PROBLEMS</td>
<td>Lackluster support for runtime handling of units</td>
<td>4</td>
</tr>
<tr>
<td>LACK OF ADOPTION</td>
<td>Cost of dependency</td>
<td>4</td>
</tr>
<tr>
<td>LACK OF ADOPTION</td>
<td>Big effort with large codebase</td>
<td>2</td>
</tr>
<tr>
<td>UOM PROBLEMS</td>
<td>Dimensional inconsistencies</td>
<td>2</td>
</tr>
</tbody>
</table>

Starting with the most frequently occurring code in the table 'bad usability', a considerable amount of frustration is evident from forum goers about the overcomplicated nature of unit libraries, specifically C++’s Boost Units library. Comments range from "ridiculously complicated" due to "insane [amounts] of templated magic going on" to "fatally flawed by the lack of good tutorial documentation" with "complicated implementation" and that to make proper use of the library, one would have to "dig deeply into it". For languages such as Java, a noteworthy comment was that the lack of operator overloading means that a UoM library
would automatically be quite "intrusive and cumbersome to use", the extra code friction will mean that it will be used less in practice. Lastly, although this might be more related to type restrictions, a good comment was that "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". Another commenter echoed this statement, saying that "you still need conversions when using general purpose mathematical libraries or reimplement them as type-generic functions which opens a whole other can of worms and is sometimes not possible in a sensible way". With regards to type restrictions, one commenter wrote "all UoM libraries fail miserably at supporting anything but simple computations", giving an example such as a linear-algebra library such as Eigen which does not allow for UoM types to be preserved across computations, which is stated as a "showstopper" for adoption of UoM libraries. Another example given by a commenter in their personal project was that they had to convert back and forth from Doubles in order to apply a linear regression from the 'statistics' package. These comments echo previous concerns from previous sections that type restrictions are still a very large concern regarding these UoM libraries, possibly due to the overcomplicated implementation/nature of UoM solutions.

More straightforward and commonly cited reasons to the lack of adoption was having a 'simple use case', with 'no gain' to be reaped from unit libraries and that there was 'no immediate need'. There are quite a few reasons and examples to pick from. One example reason was "narrow requirements" for Java projects as usually Java projects "do too little numerical calculations and with only a few units, so nobody bothers to do it strictly with units". According to some commenters, unit errors are infrequent and easy to catch during testing, another commenter believes that they do not feel like they are writing the kind of code that would benefit from it.

Towards the next code 'Precision loss', it was quite common to observe from the forum goers how libraries such as Boost C++ defaults to double for underlying storage, furthermore the library does not let developers to pick the units or scale of the backing store (e.g. lengths are always stored in meters). Accordingly, this would lead to a loss of precision if something other than doubles were to be used. Further complaints with regards to unit conversions are that they are "inaccurate and lossy". With Java (and C, C++, C#, etc...), "extra thought and design" has to be put into the floating point aspects of developer tools. As compounding loss of precision can lead to dramatically wrong results.
A lot of the commenters had their own solutions already "written inhouse" and that "Not Invented Here" syndrome was prevalent, which is the same reason "people roll their own StringUtil libraries" according to one commenter. Another commenter answered that they "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".

For the code 'Unaware', a plethora of examples are given. One example that a commenter gave was that for their simulation of fluid flow in Fortran, no one was aware of units but everyone was interested in performance. "Awareness is a major issue" was quoted and that it does not occur to people to "even try to employ units of measurement", even the distinction between dimensions and units "is not made clear" to developers, which is reflected by the whole topic. So the lack of awareness regarding both UoM solutions in addition to the general topic (units and dimensions) are apparent.

For others, using compile-time unit libraries such as Boost Units would require them to systematically 'templatise' every bit of their codebase that touches units, which is entirely a non-starter for them. They concede that using a UoM library that handles units at runtime would theoretically be a possibility, but it is nevertheless unlikely as they would be going back through a codebase of around 1 million LoC (lines of code).

With regards to performance, another note was that in even areas of applications, simulation performance is often bottlenecked (e.g. most physical and engineering calculation), the main priority thus goes towards the best numerical algorithms and implementations. Therefore, in applications that do lots of calculations, their performance tends to matter more and boxed values with types would have unnecessary overhead.

To end this section, a few more codes such as 'dimensional inconsistencies', 'cost of dependency', 'priority towards performance', 'not being paid to develop code' and 'big effort with large codebase' are taken into perspective. A very insightful comment was given regarding scientists and their code where most scientists consider code something secondary in their work; sometimes even a burden and that most scientific code is not touched anymore after they publish their article which "does nothing to contribute to the situation". Another commenter stated that the cost of dependency introduces more awkwardness at the type level if
the calculations being done are not realistic sources of bugs to defend against, preferring to rather spend that type level complexity on something else (or use simpler types). The external dependency also does not seem worth it for a commentor as there is no ’clear standout’ like commons-lang for Strings or commons-io for I/O.

Therefore, the following concerns are most highlighted in this section:

- awareness and general lack of concern, also the overcomplicated nature of some UoM libraries in some cases might deter developers away from them
- developers and scientists might have simple use cases, and there is possibly no gain to be made by having to learn a UoM library for a simple use case, especially in business areas where Java is most commonly used for example
- precision loss due to the inner workings of some UoM libraries (such as Boost C++) which defaults to using doubles for underlying storage, a concern that leads to lossiness of data
- type restrictions and tricky integration of UoM solutions with other libraries, as well as the concern over having to modify swathes of currently existing code to support UoM (cost of dependency)
- priority for performance over unit safe code, the cost of having unit safe code could come at the expense of performance, this concern is especially apparent for simulations for example, and more priority is placed for making sure simulation code works first rather than being safe from unit errors
5.3 Design Principles for Unit Libraries

It is therefore clear that the following design principles from the exploratory DSR study committed emerge:

1. developers must favour simplicity in their design of unit libraries to reduce verbosity and boilerplate code (which might mean moving away from clunky, generic style type programming by wrapping these approaches entirely), in addition to that, unit code must return meaningful error messages in case of any unit incompatibilities

2. UoM libraries must be extensible enough to be supported by all libraries, this means that UoM developers need to get together to define a unit library standard interface so that these libraries can then be supported by the wider programming community

3. allow users of UoM to modify the underlying type of storage (e.g: integer instead of UoM libraries defaulting to doubles) to combat precision loss

4. increase awareness of UoM by highlighting commonly occurring unit disasters and push for including UoM in standard libraries of main programming languages to increase awareness of UoM (which is why the F# community is aware of UoM due to the builtin UoM library in the standard F# library)

5. implement better dimensional analysis techniques as highlighted in the theory chapter as some libraries do not perform well in that regard (according to some respondents)

6. increasing social impetus and push for UoM by getting the right people to galvanise these technologies
6 Conclusion

It is not so easy to place exact reasons for the lack of adoption and general lack of awareness of UoM libraries owing to the multifaceted nature of the topic. However, as the first of its kind in terms of the research field, this thesis has given an initial insight and presents convincing reasons to the mentioned lack of adoption.

What can be said with certainty however is that there is a considerable lack of awareness surrounding UoM libraries. It has been reiterated throughout the thesis quite a few times that this is a common, reoccurring theme of the research that was carried out. It seems as if people are aware of unit failures and their implications, but what they generally do not know is that there exists libraries out there that can handle and prevent these failures to some degree. In the event that they were aware, many were simply 'not bothered about it' or perhaps even 'lazy' and generally did not feel like going out of their way to adopt a UoM library. Alternatively some found it unnecessary to add an extra dependency to a UoM library to prevent these issues (which is understandable to some extent if the use case is simple). Some were satisfied with their own solutions (due to e.g: tradition or fear of change), but as mentioned previously, there are many pitfalls with having an own solution. To increase awareness of UoM solutions from a personal point of view, unit libraries must be bundled with more general purpose programming languages as a part of their core API's, similar to Units of Measure in the F# core library (which itself, has increased awareness of UoM in F# circles based on some qualitative data points).

For others who are aware of UoM solutions, there are a variety of reasons for the lack of adoption which are vast and quite understandable in retrospect. People are generally hesitant in converting their codebases to work with UoM libraries and there is an overhead to getting started with one, there is uncertainty with database interactions. Above all, one of the most convincing reasons given is the lack of support for UoM libraries by other libraries such as linear algebra libraries (e.g: Eigen) as UoM types are not preserved. The concern for compilation time of code was also a concern as, mentioned previously, every temporary variable in a matrix multiply would have to have a unit, which drastically increases computational complexity. This is a worrying concern for UoM solutions as they are generally geared towards computation and number crunching with units, but if they are not flexible enough with maths libraries, then the gain of a UoM library would not be worth it. It is almost as if UoM libraries are good at what they do but only internally, and even so, some suffer from
being overly complex, having to write excessive boilerplate code, inability to handle units at runtime properly, bad usability and bad error messages, thus positioning themselves awkwardly in the grand scheme of things.

Echoing one of the short answer texts in the questionnaire for a respondent, development was primarily 'computer-driven', datatypes represented how the computer thinks about the data, it was only recently (for some reason) that they themselves recognised datatypes as something useful. Perhaps this is an indication that developers ought to think more about units and types than just the data to be computed on, a type of development style that naturally follows when programming in F# or Haskell.

Regarding the second research question, "what is the severity of this issue in terms of frequency and cost?", an attempt was made at quantifying it, but proved rather hard to do so on second thought as the data gathered did not signify much regarding it and remains unanswered. Perhaps a more detailed and specific study is required to answer this question as it has a different focus than the main and primary research question regarding the reasons behind the lack of adoption of unit libraries.

As for future directions in this field, hopefully the information produced in this thesis will serve as a guide for UoM developers in understanding why UoM libraries have not been adopted as much they should have. In addition to producing the steps required to encourage adoption of UoM libraries, or address the issues surrounding the specific concerns of these libraries.
References


Appendices

A Questionnaire Extra Qualitative Data

Table 8: (TTL-Y1, D-Y1 and M-Y1) collated

<table>
<thead>
<tr>
<th>Why was the reason to adopt Unit measurement libraries not considered earlier?</th>
</tr>
</thead>
<tbody>
<tr>
<td>They are very specific. More template magic is needed.</td>
</tr>
<tr>
<td>Manual conversion functions covered most common cases</td>
</tr>
<tr>
<td>In our case, it was adopted very early in the process.</td>
</tr>
<tr>
<td>Preventing bugs</td>
</tr>
<tr>
<td>Hadn’t thought of it</td>
</tr>
<tr>
<td>The only thing I use is std::chrono. And just because it is easily available.</td>
</tr>
<tr>
<td>Investment of effort required to use such a library</td>
</tr>
<tr>
<td>Unaware of the possibility</td>
</tr>
<tr>
<td>Didn’t think of it</td>
</tr>
<tr>
<td>Units of measure liquidate a huge class of mistakes in the non-trivial computations.</td>
</tr>
<tr>
<td>They were considered from the beginning because I needed to represent values in multiple unit systems.</td>
</tr>
<tr>
<td>Because their existence was not yet known to me and my coworkers at the time</td>
</tr>
<tr>
<td>Not aware</td>
</tr>
<tr>
<td>I believed that the units in my calculations would be clear.</td>
</tr>
<tr>
<td>Difficulty of using it</td>
</tr>
<tr>
<td>Working with flight simulations where units are very diverse (mixed SI/Imperial)</td>
</tr>
<tr>
<td>Reluctance to depend on external libraries for a low-priority feature. We started using them when we switched to F#</td>
</tr>
<tr>
<td>being built in to the language was key.</td>
</tr>
<tr>
<td>We don’t really have unit of measurement in our system but we do use unit of measure to differentiate ids, since</td>
</tr>
<tr>
<td>f# unit of measure doesn't have a performance impact it fit our use case</td>
</tr>
<tr>
<td>Haskell’s numeric hierarchy is incompatible with the notion.</td>
</tr>
</tbody>
</table>
Why was the reason to adopt Unit measurement libraries not considered earlier (Cont.)?

Safety in all operations involving conversion between UoMs

Lack of talent to make it accessible, and compatible with rudimentary in-house data structures.

Lack of experience.

Low usage of C++11

Lack of understanding in development teams

Hadn’t previously needed it, as our projects started moving into robotics controls it became clear it was necessary.

If all your work is in one UOM, you don’t need to worry about it initially

I think earlier on development was rather computer driven, i.e. datatypes represented how the computer thinks about the data and only more recently types get recognised as something useful.

Units of measure were a benefit to switching my project from to F# from C#. I didn’t switch because of them alone, but they have been very useful.

Perception that they’re not needed, technical complexity, interaction with DBs

Difficulty of choosing among libraries and assessing whether a particular library fits the needs.

Units of measure plugins often don’t support fractional units.

Matrices often require heterogeneous sets of measures.

None of these issues were addressed by adoption and were all a tax on subsequent use.

Difficulty of choosing among libraries and assessing whether a particular library fits the needs.

Problem didn’t require it.

Lack of understanding in development teams

Robustness against mistakes, compile time checking.

Not required.

It was considered and used from the beginning.
Table 9: \((\text{TTL-N1,D-N1,M-N1})\) collated

**Why haven’t they been adopted?**

They have all failed to represent the domain language succinctly
Too flexible.

We needed more things to be compile-time errors. We also wanted things to be backed by integers for performance.
Conversion introduces error, reduces precision for large and small models as consistent systems are generally better.
I don’t think of them, and we haven’t had units bugs
Not really applicable to our problem domain
Compilation overhead not worth the benefit of just unit-testing math code
There is no standard.

Boost.units ‘quantity<T>’ (and others) do not preserve standard-layout, triviality and podness of ‘T’, which are critical for our domain

Most developers are not familiar with it (scared by compiler errors)
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, etc.) that interact very poorly with UoM libraries. We would use them if we could.
Small set of units, little need for conversion
Client didn’t pay for it
In general code should be unitless
Template instantiation baggage is too costly for benefit
Because mixed units

Is there a modern C++14 header-only units library? That’s what we would be looking for. I must confess I haven’t looked for it so there may very well be one.

Perhaps units are too little of a concern for a let’s say "more general" codebase where many things are without units.
Not needed.

A bit of consistency in the backend and doing "inconvenient units i.e. imperial" on the UI/during IO has worked out fine.

This isn’t an important problem for me.

I find their implementation (in F#) inelegant to use and, in particular, the way they are added and removed from numerical values feels hacky.

Because mixed units
We don’t have units
No support for units that are modifiable at runtime as opposed to compile time.

Inertia
Why haven’t they been adopted (Cont.)?

It was considered, and dismissed.

Code is relying heavily on C++ templates and converting everything is too costly.

I am not sure it wasn’t considered. I think it isn’t much of a problem.

Lack of awareness

Why should they be?

Programmers don’t think they are necessary until someone makes a mistake

Not needed

No need

Custom solution

Lack of knowledge that this is a problem and/or value of using them not perceived better than pain of learning a new library.

Mistrusting the accuracy, using custom functions that would improve it.

Today is the first day i’ve ever heard of them, might be because i live under a rock but after learning of these i likely still won’t be using them as i like to make stuff as personal and hand tailored to my current needs as possible.

Units are standardised, and all conversions take place with hand rolled inline equations.

Not part of std libraries for the most part

I didn’t even know there were libraries that did this for you

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.

Not known

Too much friction using with non-UoM libraries

Lack of urgent need

No-effort “sweet spot” solution not easily found

I’ve never felt a need for them

I’m not aware of an approach that’s good enough on balance of extra safety vs convenience. 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.
Why haven’t they been adopted (Cont.)?

They cause more pain than they relieve

We don’t deal with large amounts of numeric data, don’t do many operations involving multiple types of unit, and when we do they’re very different types of units that measure very different things, and we don’t use different units to measure the same thing. Almost every numeric instance variable would have its own unique unit.

Adding a UoM framework would add extra intellectual encumbrance that in the end would likely cause more bugs than it would help prevent.

The libraries always fail to interact with the rest of the ecosystem.

Unnecessary in our domain

Because we’re not used of it

Not seen as a crucial feature, more of a "nice-to-have"

No support for units that are modifiable at runtime as opposed to compile time.

Perceived effort cost of porting existing, 'working' code.

When you use a consistent set of units in your calculations, there isn’t much of a problem.

I know nothing about them

Not available as standard in the platform
## Table of Codes

Table 10: Codes under category Lack of Adoption with frequency of occurrence of codes

<table>
<thead>
<tr>
<th>Code Number</th>
<th>LACK OF ADOPTION</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Not a crucial feature</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>Client didn’t pay for it</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Don’t care</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>Lack of experience</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>Lack of talent</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>Cost of dependency</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>Hesitance</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>Simple use case</td>
<td>14</td>
</tr>
<tr>
<td>9</td>
<td>No gain</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>Costly to adopt</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>Priority towards performance</td>
<td>5</td>
</tr>
<tr>
<td>12</td>
<td>Not being paid to develop code</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>Licenses</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>Big effort with large codebase</td>
<td>6</td>
</tr>
<tr>
<td>15</td>
<td>Disagreement over standards</td>
<td>7</td>
</tr>
<tr>
<td>16</td>
<td>Don’t care about standards</td>
<td>3</td>
</tr>
<tr>
<td>17</td>
<td>Have/Create their own solution</td>
<td>14</td>
</tr>
<tr>
<td>18</td>
<td>Lack of social push</td>
<td>4</td>
</tr>
<tr>
<td>19</td>
<td>No immediate need</td>
<td>22</td>
</tr>
<tr>
<td>20</td>
<td>Overhead to getting started</td>
<td>6</td>
</tr>
<tr>
<td>21</td>
<td>Unaware</td>
<td>22</td>
</tr>
</tbody>
</table>
### Table 11: Codes under category UoM problems with frequency of occurrence of codes

<table>
<thead>
<tr>
<th>Code Number</th>
<th>UOM PROBLEMS</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Non-idiomatic</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Inelegant/Hacky</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Not good enough</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>Framework lacks support</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>Lackluster support for runtime handling of units</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>Tedious/Verbose</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>Dimensional inconsistencies</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>Bad usability</td>
<td>13</td>
</tr>
<tr>
<td>9</td>
<td>Bad error messages</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>Bad implementations</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>Bad integration</td>
<td>6</td>
</tr>
<tr>
<td>12</td>
<td>Functions incorrectly</td>
<td>7</td>
</tr>
<tr>
<td>13</td>
<td>Lacks support for fractional units</td>
<td>3</td>
</tr>
<tr>
<td>14</td>
<td>DB interaction</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>LANGUAGE SPECIFIC</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>Precision loss</td>
<td>8</td>
</tr>
<tr>
<td>17</td>
<td>Type restrictions</td>
<td>8</td>
</tr>
</tbody>
</table>

### Table 12: Codes under category Different communities and unit sets with frequency of occurrence of codes

<table>
<thead>
<tr>
<th>Code Number</th>
<th>DIFFERENT COMMUNITIES</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Tradition</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>Divergence</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Develop themselves within it</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>Different intuition for different unit sets</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>Different sets of units</td>
<td>5</td>
</tr>
</tbody>
</table>
### Table 13: Codes under category Successful implementations with frequency of occurrence of codes

<table>
<thead>
<tr>
<th>Code Number</th>
<th>SUCCESSFUL IMPLEMENTATIONS</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Brazilian communities</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>Deutsche telecom smartphone</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Siemens container project</td>
<td>1</td>
</tr>
</tbody>
</table>

### Table 14: Codes under category UoM desired features with frequency of occurrence of codes

<table>
<thead>
<tr>
<th>Code Number</th>
<th>UOM DESIRED FEATURES</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Good error messages and tooling</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Type flexible</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>Easy integration and dimensional consistency</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>Good usability</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>Idiomatic</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>Lightweight</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>Standard</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>Support runtime unit and handling</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>Wrapper solution</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>Support custom units</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>No need to rewrite existing software</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>Intuitive</td>
<td>9</td>
</tr>
<tr>
<td>13</td>
<td>Handle enormous dimensions</td>
<td>9</td>
</tr>
</tbody>
</table>