MEng Civil and Environmental Engineering Final Year Project

Functional Dimensional Analysis for Engineers Using Haskell

Jaimin L. Symonds Patel

June 15, 2023

Abstract

Just as the tools we use determine the efficacy of what we build, so too do programming languages determine the structures of the programs and solutions we make. The imperative style of programming is frequently used to tackle engineering problems, and fosters the same style of thinking for tackling problems. In this paper, we show what the functional programming paradigm can offer through the use of the Haskell programming language to carry out two aspects of dimensional analysis. Firstly, Haskell has been used to produce a structured program via modularisation of function composition to find the range of sets of Buckingham Π terms through the method of repeating variables. Secondly, a structured program via modularisation of algebraic data types has been produced to provide a means of uncertainty propagation through Buckingham Π terms. We conclude that Haskell can offer unique structures to clearly and concisely characterise a problem, and has at least done so in a useful way for dimensional analysis, as in this paper.

1 Introduction

1.1 Aims and Objectives

This paper explores the utility of the Haskell programming language in tackling problems within dimensional analysis. Frequently used in engineering, dimensional analysis ultimately is a method whereby a phenomenon is dissected based on the sole premise that it can be described by a dimensionally consistent equation of certain variables (Zohuri, 2015). Indeed, dimensional analysis is therefore incredibly general, which means that working with such a method of analysis can become rather abstract, and seem like it may lack structure or tangibility.

It is precisely this seeming lack of structure that warrants the testing of Haskell on such a problem. Haskell is a very unique programming language, primarily because it is one that is purely functional. This means that every structure made in Haskell can be represented by a mathematically pure function, with one or more inputs, and an output. There are other key aspects of Haskell explained in Section 2, but needless to say Haskell's unique features allow it to express structures in ways that imperative paradigm languages struggle to express.

Now, this paper is not suggesting at all that Haskell provides the best toolkit to solve the particular problem described in this paper, much less that it is a silver bullet for all problems within engineering. That being said however, it seems that Haskell is certainly late to the engineering table as it were. Fundamentally imperative programming languages have been commonly used in universities and in industry to tackle engineering problems for many years, such as Python and MATLAB to name a few. This is by no means a drawback, however it begs the question whether there are programming languages better suited to tackle certain problems. It is a given that there is no single best programming language that can help solve all problems, however it then seems contradictory to rely on a few similar languages as if they were the best. Therefore, there should be no reason why we would not look toward Haskell as a useful tool, if it seemed to tackle a problem in a clear and concise manner.

With that said, it is the aim of this paper to show that Haskell is capable of solving at least the problems in dimensional analysis presented here, in a structured and representative way. The specific problems presented in this paper amount to two aspects of what is fundamentally the same problem. Those two areas are the finding of Buckingham Π (Pi) terms, and uncertainty propagation via those same Buckingham Π terms. The former area focuses on finding dimensionless quantities (called "Π terms") from a list of physical variables known to be related to a physical phenomenon (an example would be $\Pi' = st/d$ for speed, s, distance, d, and time, t, where the unit of Π' here is subsequently dimensionless). The latter area focuses on finding resultant probability distributions of those Π terms when the variables they contain are realised to be known random variables.

The only reason why these two areas are treated separately is because their current implementations in Haskell hold very different structures. This means that for now at least, they are better treated as separate computations.

It is worth noting that at the time of writing, no such Haskell code or libraries were found online that tackled the specified problems in this paper. It is therefore an indirect aim of this paper that others might be able to build on this work, to improve the economy of Haskell's known abilities.

As a final note here, this paper cannot possibly seek to teach all of Haskell whilst seeking the aims already outlined above. Therefore, even though source code snippets are provided for transparency and clarity, it is not expected that the reader would understand every detail that is within the code. However, this paper seeks to show primarily the higher level structures of the code, from which an evaluation of Haskell's performance can be made. Thus, it is only expected that this higher level of understanding may be made when viewing the code, along with its explanations. For those who have not used Haskell at all, it is hoped that Section 2 may provide a sound enough basis to understand the common abilities of Haskell.

1.2 The Functional Programming Paradigm and Haskell

Before moving on to the particular problems that Haskell has been implemented upon, a brief note on Haskell and the functional programming paradigm must be made. Haskell is one of many functional programming languages. However, it is the primary language noted for its strict adherence to purely functional programming principles. Haskell is statically typed, purely functional, and also has lazy evaluation (Allen et al., 2016). A statically typed language is one where types of data structures must be made explicit. Lazy evaluation is the capability of the language to evaluate computations only when they are needed. The rest of this section will focus on what it means for Haskell to be a purely functional language.

For those unfamiliar with programming paradigms, it would help to first visit the one most are familiar with, but may not known it: the imperative programming paradigm. This paradigm is defined as the way of programming in which the programmer directly manipulates the memory state of a computer. For example, incrementing a variable or jumping to a point in memory are examples of changing the memory state of a computer. This is the most common way of programming, probably because it is directly linked to how the computer actually functions.

The functional programming paradigm on the other hand is defined as the way of programming in which the most basic operation is the application of functions. Thus, everything is programmed in a way that involves the application of functions, in the ordinary mathematical sense (Hughes, 1989). Functional programs do not contain assignments in the strict sense, since variables are immutable (they do not vary once given a value). Functional programs furthermore have no side-effects in any way, in the sense that a function call can have no other effect than compute its result. As Hughes (1989) says, this gets rid of a lot of sources of bugs, and the flow of control of a program is only defined by the sequencing of function outputs. Overall, functional programming has no mutable memory, no side-effects, and a limited flow of control.

Hughes (1989) is right to say that all of these things sound like constraints. But in engineering, constraints play a key and often liberating role. It is precisely these constraints that allow programs to be made in a structured manner. What then actually makes a program structured is that it is done in a modular way (Hughes, 1989). Getting rid of mutable memory forces us to not use interdependent variables, and just focus on what each function needs as input and how outputs can be strung together as inputs to other functions. Getting rid of side-effects forces us to streamline all functions to have clear inputs and outputs no matter what action they are performing. Finally, a limited flow of control forces us to focus purely on how functions are composed. All in all, these "constraints" force us to be modular.

It seems that the imperative paradigm forces us to start with the low level details of an algorithm, before structuring it as an overall picture. And in a like manner, the functional paradigm forces us to first get the overall flow and representation of data sorted before any of the algorithmic details. Furthermore, the imperative paradigm seems to be modular only in the measure that it adopts pseudo-functional principles, especially through high level libraries. When thought about, it can be said that typically imperative languages are most useful and pleasant when we do not have to worry about manipulating memory, or what side-effects functions may have, or what the specific order of computation is. This is precisely what the functional paradigm offers, and what it was built for.

It is this modular design, no matter which language is used (although with some languages being more easily suited to such design), which enables improvements in productivity (Hughes, 1989). Small modules can be coded easily, general purpose modules can be reused, and modules can be tested independently, which decreases effort debugging. Hopefully, the seeds of such benefits may be seen throughout this paper, whether in full, or just in part.

2 What Haskell Looks Like

Since Haskell has been used in this paper to reason about processes and structures, it is first necessary to give a brief walk through on the features and syntax of Haskell. This is also necessary because of Haskell's uniqueness and rarity of being used within the engineering field.

2.1 Types

The first, and possibly most important concept within Haskell for anyone wishing to use it, is that of types. We can know about types from other programming languages, but more and more frequently languages are being used in a dynamically typed manner. This means the user rarely has to make a data type explicit in most modern programming languages, but you have the option to make a data type explicit in relation to a variable.

But Haskell is statically typed, which requires that types are made explicit. For example, we can write

```
1 x :: Int
2 x = 5
3
4 y :: Double
5 y = 4.5
```

where `x` would be a fixed precision integer type, and `y` would be a double precision floating point value. The `::` symbol represents a type assignment for a single value, or a function. There are some other important types, as can be seen in

```
1  a :: Char
2  a = 'd'
3
4  b :: [Int]
5  b = [1,2,3,4,5]
6
7  c :: String
8  c = "hello" --(== ['h','e','l','l','o'])
9
10 d :: (Int,String,Double)
11 d = (5,"hello",2.467)
12
13 e :: [(Int,Char)]
14 e = [(6,'e'),(3,'g'),(2,'h')]
15
16 f = Bool
17 f = True
```

where the value `a` above is of the type `Char`, which represents a character. The value `b` is more interesting, and is of type "list of fixed precision integers (`Int`)". In Haskell, a list can contain any type, including more lists, but the catch is that it can only contain values of a single type. A list is denoted by square brackets, and furthermore, the value `[]` denotes an empty list. The value `c` is of the type `String`, which is fundamentally a list of `Char` values (note that comments in Haskell are denoted by "`--`").

The value `d` is of the type "tuple of an `Int`, a `String`, and a `Double`". A tuple is like a list, but it contains a definite number of elements (we can see in the type declaration that `d :: (Int,String,Double)`, which specifies a tuple of three elements specifically). Furthermore, it can contain elements of various types, but these have to be specified for whichever element those types occupy. When used, tuples are usually two to five elements long, and tuples any larger would warrant the use of another data type. The value `e` just shows that we can construct more and more complicated values, and as such, is of the type "list of tuples, each containing precisely one `Int`, and one `Char`". The final value `f`, is of the type `Bool`, that is, Boolean. Hence, it could either be `True` or `False`.

There are far more types which Haskell has, some of which will be touched on later when we visit type classes. This whole economy of types and structures within Haskell is called the "type system" of Haskell.

2.2 Pure Functions

Since Haskell is a functional programming language, one could say that pure functions are one of the most important aspects of it. The first thing to appreciate is that a purely functional programming language is not based on the standard model of computation, which is the Turing machine, but is based on what is called lambda calculus (Allen et al., 2016). Going into more detail would be beyond the scope of this introduction to Haskell, but in a nutshell lambda calculus can be thought of as a more formalised representation of computation, that simply uses function application as its primary operation (Aaby, 1998).

With that in mind, Haskell represents values and functions in a far more mathematically pure form than in fundamentally imperative languages (like C, Fortran, MATLAB, and Python). If we write $y = 2$ in a general mathematical context, it makes no sense for it to become $y = 4$ two lines further down. And so, in like manner Haskell treats all values as being immutable. Furthermore, just as in mathematics, functions are always pure (they simply take one or more inputs and give one output). In fundamentally imperative languages, functions can do anything on the way to giving an output, if they give an output at all (those things which they may do on the way are called "side-effects"). An example of a side-effect would be a function that only adds one to a number, and then prints the result to the screen. Technically, printing the number to the screen does not constitute an output, even though it may seem like it, since it returns nothing to whatever called that function. Hence, this would be a side effect. Haskell therefore has none of these side effects, but makes everything structured in a purely functional way (Jones & Wadler (1993), Launchbury & Jones (1995)).

Going forward then, Haskell gives types also to functions, so we can write

```
1 add1 :: Int -> Int
2 add1 x = x + 1
```

where the type of `add1` is a function that takes an `Int`, and returns an `Int`. Furthermore, `add1`'s definition, below the type declaration, tells us what it actually does when it takes an `Int`, `x`, and returns it (adding one to whatever `x` was). And so, if we were to run this function by giving it a value of type `Int`, in an interactive environment such as GHCi (which is akin to Python's interpreter or MATLAB's command line interface), then we would write at the GHCi prompt

```
1 GHCi> add1 5
2 6
```

where `5` is the input to the function `add1`, but could be any number (that is an integer). Function application also requires no brackets, but just a space between the function and the input. Notice that we did not specify the type of `5`. This is an instance where Haskell is able to infer the type, since it knows that the function can only take an `Int`. Relying on Haskell's type inference system too much though is usually considered bad practice since it takes away the clarity of explicit types, and usually results in a type error for larger programs (what is called a "type mismatch"). If we were to write

```
1 GHCi> add1 5.0
```

then we would get an error, because "5.0" is in a form that could only be a floating point number, not an Int. Therefore, Haskell complains that the input is not what add1 expects. If we wanted to be more rigorous, we could have written

```
GHCi> add1 (5 :: Int)
```

where we are explicitly stating that 5 is of the type Int.

We can also write functions with more than one input:

```
add2n :: Int -> Int -> Int
add2n x y = x + y
```

where the type declaration states that add2n has the type of a function which takes in two values of type Int, and returns a final value of type Int (which is the sum of the two numbers it takes as input). For functions of more that two inputs, this pattern of type declaration is simply repeated. Again, function application is denoted by placing a space between the inputs and the function, such as in

```
GHCi> add2n 5 4
9
```

where Haskell infers that 5 and 4 must be values of type Int.

Within Haskell's interactive environment, GHCi, we can also inspect the type signatures of values and functions using the :t command. The reason that this is important is because there are several functions that Haskell has predefined, such as addition, which we usually need to inspect to check their types. Thus, we can inspect the addition function, (+), by writing

```
GHCi> :t (+)
```

where the brackets are just used to denote that it is an infix function (meaning it usually goes between its two inputs). If we type the above, GHCi would give the type declaration of the addition function, (+), which would look something like

```
(+) :: Int -> Int -> Int
```

where GHCi would actually add some more information, which we will touch on later on when we talk about type classes.

Finally, here would be a good time to also introduce the function composition operator, (.). This is an infix function (like (+)) that has the type

```
GHCi> :t (.)
(.) :: (b -> c) -> (a -> b) -> a -> c
```

where (b -> c) and (a -> b) are both functions that take only one argument. The output can be thought of as another function, but given a value of type a as well as the first two functions, the output is a value of type c (the output of the first function passed as an argument to (.)). Hence, if we were to write (add1 . add1) 4 in GHCi, we would then get

```
GHCi> (add1 . add1) 4
6
```

where the function composition operator is composing the function add1 to another add1 (resulting in the addition of 2). Note that the order of function composition is from right to left. So, if

we had a function `f`, and another `g`, then `(f . g)` would mean "do `g` and then `f`". Furthermore, there is another function, "`($)`", which is the function application operator (also infix). This has the type

```
GHCi> :t ($)
($) :: (a -> b) -> a -> b
```

where `(a -> b)` is a function of one argument, and `a` is the type of a value which is the input to that function, and the result is of type `b` (the type of the output of the function). In other words, `add1 3` is equivalent to `add1 $ 3`. This may seem pointless, but with longer expressions, instead of

```
add1 (add1 (add1 (add1 4)))
```

we can write

```
add1 . add1 . add1 . add1 $ 4
```

which is far more readable as the brackets not being needed since `(.)` and `($)` make the order of function application explicit for us.

2.3 Functions Being First-Class

This a more subtle point, but functions being first-class means that Haskell treats functions as first-class citizens as it were. This means functions can be passed to other functions as inputs, they can be returned as the output of other functions, and they can be stored in data structures as values in their own right (Allen et al., 2016). An example of this is found by using the predefined function `map`, where it has a type defined as

```
GHCi> :t map
map :: (a -> b) -> [a] -> [b]
```

where `(a -> b)` is a function that takes an input of type `a` (any type), and returns an output of type `b` (also of any type). And so, `map` takes 2 arguments, the first being a function of one argument, and the second being a list containing elements of the same type as `a` in the function. With this type declaration, it might not take too much effort to realise then that the function `map` applies the function it receives to each element of the list which it also receives (hence it *maps* the function onto the elements of a list). The variables `a` and `b` in the type declaration are known as "type variables", which can represent either a specific range of types, or any type (which is the case here). Here they simply state that whatever type `a` and `b` are when `map` actually receives defined arguments, those are the same types in all instances where `a` and `b` occur. For example, if we give `map` our previous function `add1`, and a list of numbers, the following would happen:

```
GHCi> map add1 [1,2,3]
[2,3,4]
```

Therefore, for this specific application, the type signature of `map` would essentially be `(Int -> Int) -> [Int] -> [Int]`.

Furthermore, we can put functions in lists and treat them as values, such as in

```
GHCi> funcList = [add1,add1,add1]
GHCi> :t funcList
funcList :: [Int -> Int]
```

where `[Int -> Int]` means "a list of functions that take an `Int` and give back an `Int`".

2.4 Recursion and Pattern Matching

Since Haskell's data structures are immutable, there cannot be any iterators or values which change value upon iterations. But what Haskell can do, in light of it being purely functional, is carry out recursion. In a sense, again, this is a more mathematically formal representation of iteration. There are a few things to note, but we can write

```
fact :: Int -> Int
fact 1 = 1
fact n = n * fact (n-1)
```

where we have "pattern matched" the function `fact` (a function to calculate the factorial of a number) to do certain things for certain values. So, when `fact` is called with the number `1` as the input, it will give `1` as the output. But, for any other number, it will take that number and multiply it with the `fact` of that number minus one. For example, if we had

```
GHCi> fact 3
```

then Haskell would look at our function definition and see first if our input to `fact` is `1`. It would see that `3` is not equal to `1`, and then move on to the next pattern in our definition. Since `n` represents any number, it would catch anything that is not `1`, and so Haskell would follow through with our second pattern to get

```
3 * fact 2
```

where this time when we call `fact` again (recurse), the value `2` passed to `fact` matches the second pattern, giving

```
3 * 2 * fact 1
```

where afterwards, things change. `fact 1` actually matches our first pattern in our definition for `fact`, which Haskell looks at first when it evaluates `fact`. Therefore, the particular case, `fact 1 = 1`, is called the "base case" since it finalises the expression to give

```
3 * 2 * 1
```

which can be evaluated to `6`.

Thus, in Haskell, there are different (though equally powerful) "tools of the trade" when compared to fundamentally imperative languages. And so, instead of for or while loops, Haskell has recursion and pattern matching.

2.5 Algebraic Data Types

Another core feature of Haskell is not only the predefined set of types which it has, but also the capability to easily allow us to define our own types. Another simpler capability is that we can use type synonyms to refer to predefined types. Consider a function that takes a date (day, month, then year), and returns the same date with a year added onto it. For this, we might write

```
add1year :: (Int,Int,Int) -> (Int,Int,Int)
add1year (d,m,y) = (d,m,y+1)
```

which carries out the correct computation, however lacks information which could be more helpful. Let us say we wanted to inspect this function because we forgot how it was defined, and we wrote

```
GHCI> :t add1year
```

then we would get

```
add1year :: (Int, Int, Int) -> (Int, Int, Int)
```

which only tells us so much. We cannot know what the order of representation of the date is, nor can we really be certain that each tuple containing the three numbers is at all a date. So then, we could make some helpful type synonyms to begin with, such as

```
type Day = Int
type Month = Int
type Year = Int
```

which allows our definition become

```
add1year :: (Day,Month,Year) -> (Day,Month,Year)
add1year (d,m,y) = (d,m,y+1)
```

where `Day`, `Month`, and `Year` are equivalent to `Int`. And so, if we inspected `add1year` again, we would get

```
GHCi> :t add1year
add1year :: (Day, Month, Year) -> (Day, Month, Year)
```

which ought to be a lot clearer. If we wanted things to be more concise, we could have written

```
type Date = (Int,Int.Int)
```

so that the type declaration of `add1year` becomes

```
add1year :: Date -> Date
```

which makes it clear what sort of information is being passed around.

This is just one example of making code a bit more understandable through playing with types. Another way is to make our own data types by using the `data` keyword. For example, using the idea of an `add1year` function like before, we could write

```
type Date = (Int,Int.Int)
data Maybe a = Just a | Nothing

add1year :: Date -> Maybe Date
add1year (d,m,y) = if (d > 31 || d < 1 )
                   then Nothing
                   else if (m > 12 || m < 1 )
                        then Nothing
                        else Just (d,m,y+1)
```

where `data Maybe a = Just a | Nothing` just means that we have a new data type called `Maybe a`. The type `a` could be anything, so for example we could have a value of type `Maybe Int`. `Just a | Nothing` tells us that for a type such as `Maybe Int`, we could have possible values `Just` (*an `Int` value*), or (which is what the "`|`" means in a `data` declaration) `Nothing`. So, for example, a value of type `Maybe Int` could be `Just 4`, `Just 2`, `Nothing` or `Just` *(any integer)*.

This helps us in our example above since we know that the number for a day cannot be below 0, or above 31. Furthermore, we know that a month cannot be below 0, or above 12. Therefore, we can use a nested if statement to catch these exceptional cases, and then return a value of `Nothing` if they occur. Note that `(||)` is the logical or function in Haskell. If, however, none of these exceptional conditions are triggered, then the function returns the date with the year increased by 1, but with that date wrapped in the `Just` data constructor so that the returned type agrees with the `Maybe Int` we put in the definition of `add1year`. Hence, a data type like `Maybe a` can be useful to sift out meaningless inputs to functions. This is just one example of the utility of custom algebraic data types.

A note on the syntax would be good however, since nested if statements can be hard to read. We can write instead

```
1 add1year :: Date -> Maybe Date
2 add1year (d,m,y)
3   | (d > 31 || d < 1 ) = Nothing
4   | (m > 12 || m < 1 ) = Nothing
5   | otherwise = Just (d,m,y+1)
```

where each Boolean statement after the pipeline ("`|`") is a conditional trigger, and the final trigger, `otherwise`, is one that catches any other possibility. It is also worth noting that the `Maybe a` data type is actually predefined in the Prelude Haskell library (which is what gets loaded automatically when Haskell is used), since it is such a useful construction.

2.6 Type Classes and Functors

This final feature of type classes will only be explained enough for a usable understanding to be gained. This is because any technical understanding is beyond the scope of this paper, and is not necessary, but it is good to know that this explanation is not the fullest of pictures.

First of all, a type class is simply the name of a group of types with shared characteristics. The characteristics in question are simply a collection of functions which can be implemented on all the types within that type class (Allen et al., 2016). For example, in Haskell there is the predefined `Num` type class, which includes all types which can represent numbers. However, more accurately, the `Num` type class contains all the types which can implement addition, subtraction, multiplication, and division, among a few other qualifying functions. After all, for something to be a number, it has to be able to be added to, subtracted from, multiplied, or divided. Thus, the `Num` type class included the `Int` type, `Float` type, `Double` type, and other data types that represent numbers.

Now, beforehand it was mentioned that we would revisit the type signature of the function, `(+)`, since `(+) :: Int -> Int -> Int` was a simplification. What GHCi actually gives is therefore

```
1 (+) :: Num a => a -> a -> a
```

where we can see the `Num` type class being mentioned. This essentially means that the addition function can be used on any type, `a`, as long as it is part of the `Num` type class (`Num a =>`

means "where the type `a` is part of the `Num` type class). Thus, "`Num a =>`" is known as a "type constraint", since it constrains the types that a function can take.

There are several type classes used in Haskell that are useful, however there is one in particular that is used strongly in this paper. The type class in question is the `Functor` type class. Bhargava (2013) gives a simple explanation of what a functor is, and simply put it is a contextual container. The equivalent explanation would be to say that the characteristic function common to all types in the `Functor` type class is `fmap`. In fact, we have already seen `fmap`, under the guise of `map`, which applies a function to all elements in a list. In the same way that `map` works, `fmap` takes a value which is surrounded by a contextual container, and applies a function to only the value, without effecting the container. Hence, `fmap` does the same thing as `map` when applied to a list, but only `fmap` can work with other containers (functors) that are not lists. Likewise, as with the reasoning for the `Num` type class, a functor (container) can only be a container if a function can be applied to what it contains (`fmap`). So, here too, we can think of functors in terms of a unique property, or the unique function that applies to them.

Another functor which we have seen is the `Maybe` functor (note that this is not `Maybe a`, since a functor is only the container, not the data type in its entirety). The `Maybe` functor can be considered a functor since it essentially packages another value within itself (such as in the value `Just 5`, where `5` is packaged within `Just`). Hence, using our previous `add1` function, we can write

```
GHCi> fmap add1 (Just 1)
Just 2
GHCi> fmap add1 (Nothing)
Nothing
```

where `Nothing` is treated as a container without a value. In the same way we could write

```
GHCi> fmap add1 [1,2,3]
[2,3,4]
GHCi> fmap add1 []
[]
```

where the empty list this time is treated as a container without a value. Since `fmap` only affects the values inside a container (functor), the container itself is left unchanged.

There are more type classes that are commonly used, such as `Show` (containing all types which can be printed to a screen), `Eq` (containing all types which can be tested for equality (or that can implement the `(==)` function)), `Ord` (containing all types that can be ordered), and `Fractional` (containing all types which are non-integer numbers), as well as more exotic type classes like `Monad`, which is definitely beyond the scope of this paper, but is infamous for its capabilities.

Because this can only be a brief explanation of the features of Haskell, it would be of great interest for those who may still have questions, to consult a further resource on Haskell[1]. However, this introduction to Haskell should be enough to understand how Haskell has been implemented in this paper, which shall now be focused upon.

[1] A very well written and concise guide to Haskell can be found at https://en.wikibooks.org/wiki/Haskell.

3 Dimensional Analysis and the Buckingham Pi Theorem

3.1 The Buckingham Pi Theorem

Barenblatt (1996) states that the idea on which dimensional analysis is based on is fundamentally simple. That idea is the fact that physical laws do not depend on arbitrarily chosen basic units of measurement. From this, the mathematical functions of such laws must also possess the property of general homogeneity, or symmetry. This finally allows one to reduce the arguments to these functions, making it easier to obtain a full function via experimentation (more on this in the next part).

Barenblatt (1996) gives also an insightful introduction to dimensional analysis, from which a parallel example will be derived here to illumine the topic in more clarity (this example will also be used throughout this paper). Let us consider the same example as in Barenblatt (1996), but in hopefully a more explicit and extended nature. And so, let us say that we wish to work out how the period of a pendulum depends on other properties. Firstly we must consider the physical variables which the period could depend upon. Such variables which the period τ depends on are the length of the rod/string l, the mass of the bob m, the gravitational acceleration g, and the initial amplitude θ_0.

In terms of practical and physical meaning, these variables are simply numbers obtained via measuring the respective quantities in arbitrarily agreed standards (Barenblatt, 1996). In other words, for obtaining the length of the rod/string for example, the length is measured by comparing it to a unit of length. It would be good to remember here still, all things mentioned are simply numbers. And so, when considering a transformation of units for each dimension, say a decrease in size of the unit, then each measurement would increase by that same factor. In the example of the pendulum, the measurement of the length of the rod/string would increase by the factor L, the mass by the factor M, and the gravitational acceleration by the factor LT^{-2} in the transformation of fundamental units whereby each dimension's units decrease by a factor of M, L, and T. Note that the measurement of the initial amplitude (measured in radians) would not change if the fundamental units are transformed, since this angular measurement is a ratio of lengths (which would increase the measured value by a factor of $L/L = 1$).

Now, these factors are all just positive numbers, and can be chosen at will. Hence, one can scale such units respectively from metres to millimetres, kilograms to grams, and seconds to milliseconds ($L = M = T = 1000$). Thus, the numerical values of all lengths, masses, and times will increase by the same factor (1000 in this case).

Barenblatt (1996) gives a key definition of the dimension of a quantity as being precisely that which determines the factor giving the magnitude of its change under a fundamental transformation in units. Thus, length has dimension L precisely because, if the unit of length is decreased by a factor of L, the numerical values of all lengths are increased by a factor L. In the same way, so too do mass and time have dimensions M and T. Nonetheless, Barenblatt (1996) makes clear that L, M, and T are nothing but abstract positive numbers. And so, the dimension for gravitational acceleration can be said to be a function of these numbers. Here it must be noted that all functions of such factors are always power-law monomials, for physical variables under consideration. Thus, we only ever observe quantities which have dimensions like LT^2, whereas we never observe a quantity that has a polynomial dimension like $\left(L + \frac{L^2}{M}\right)$. This further alludes to the fact that there are no distinguished units as it were, since all unit systems can be expressed mutually (self-similarity).

And so, here it is good to introduce those quantities that do not vary under a fundamental change of units. One example has already been met, that being θ_0. Another is that of the ratio $\tau/\sqrt{l/g}$, which one may double check. And so, there would be what are termed two "Π terms" according to the Buckingham Π Theorem, being

$$\Pi_1 = \frac{\tau}{\sqrt{l/g}} \qquad \text{and} \qquad \Pi_2 = \theta_0. \tag{1}$$

As Sonin (2001) describes, the functional relationship between physical variables $F(q_1, q_2, ...q_n) = 0$ (equivalent to $q_1 = f(q_2, ...q_n)$), can be reduced to the relationship $\Phi(\Pi_1, \Pi_2, ...\Pi_{n-k}) = 0$ (equivalent to $\Pi_1 = \phi(\Pi_2, ...\Pi_{n-k})$), where k is the number of fundamental dimensions required to form the n physical variables.

By drawing from this reduction, the physical functional relationship of variables in the pendulum example can be written as

$$\Pi_1 = \phi(\Pi_2) \quad \implies \quad \frac{\tau}{\sqrt{l/g}} = \phi(\theta_0), \tag{2}$$

and thus for the period of the pendulum we have

$$\tau = \phi(\theta_0)\sqrt{\frac{l}{g}}\ . \tag{3}$$

This equation is somewhat different to that which may be familiar, that being

$$\tau = 2\pi\sqrt{\frac{l}{g}}\ . \tag{4}$$

Of course, the difference here is that what was a function of θ_0 in Equation 3, has become 2π in Equation 4 (since ϕ would not have any other Π term as its argument, turning it into a constant). This has to do with the initial physical variables chosen right at the beginning. Sonin (2001) explains that if any independent physical variables are missing, the dimensional analysis will be erroneous. Conversely, Sonin (2001) states that a myriad of independent physical variables robs the analysis of its power, essentially by complicating the result.

That latter effect is precisely what can be seen in Equation 3, whereby the period is taken to be dependent on the initial amplitude. Now, this assumption is not wrong (Singh et al. (2018) shows how larger initial amplitudes increase the period of oscillation), but if one only cares about small amplitudes of oscillation, it does complicate things. Thus, if Equation 3 was taken as the result, a functional relationship between the original Π_1 and Π_2 terms would have to be ascertained, as opposed to Π_1 being simply equal to a constant (2π) by initially ignoring the dependence on θ_0.

But let no one forget, in the original case of counting θ_0, this example has shown how a function of four physical variables has been reduced to a function of one variable. And by not counting θ_0, a function of three physical variables has been reduced to a relationship with a constant. Now that the theory has been laid out, the next endeavour is actually obtaining the specific Π terms from a general list of physical variables. However, why this is actually important shall be touched upon first.

3.2 Motivation

Karam & Saad (2021) put succinctly the importance of dimensional analysis, as allowing one to reduce the number of control variables in an experiment. They give the more fitting example of

wanting to measure the drag force F, on a cylinder with diameter d, placed in a fluid of density ρ and viscosity μ, that is moving at a velocity v. Thus, a mathematical description that gives a functional dependence of F on the other variables gives itself as $F = f(d, \rho, \mu, v)$. Now, if one were to blindly determine this relationship via experimentation, then one must vary a single variable and hold all others constant to discover the effects on the response surface of F. All in all, this would necessitate $5^4 = 625$ experiments. However, by utilising dimensional analysis, the dependence of the problem reduces from 5 variables to 2 dimensionless Pi groups, requiring only 5 experiments for them to be determined, since $\Pi_1 = \phi(\Pi_2)$.

Furthermore, Karam & Saad (2021) describe another key use of dimensional analysis in similarity analysis between a laboratory setup with a scaled-down model and a true scale in reality. In such a use, Π terms can be used as a mapping between the two different scales, as these quantities are dimensionless and are thus held to be the same for both the real scenario and scale model.

Both of these key uses of dimensional analysis reduce the cost of experimentation by reducing the required number of experiments, and allowing for the scaling down of models of the phenomena to be measured (reducing the size and cost of equipment).

3.3 The Method of Repeating Variables

A single method called the method of repeating variables (Karam & Saad, 2021) was used in this paper for obtaining the Π terms defined by the Buckingham Π theorem. This method (outlined in detail below) fundamentally employs linear algebra such that a dimensional matrix is first formed, and then the kernel vectors of that matrix are found, as detailed by Zohuri (2015). Each resulting kernel vector contain the powers of the physical variables (denoted within the dimensional matrix) that yield a dimensionless Π term, when those variables are strung together as a monomial. For any given dimensional matrix, a set of Π terms is obtained if there are more than one linearly independent kernel vectors (where each one is a basis vector of the null space of the dimensional matrix). There is more information in this later on.

The key ability of this method is that it is able to produce interdependent non-duplicate sets of Π terms. Zohuri (2015) gives some light on these sets of Π terms, saying that there is no unique or independent set of Π terms, and that any one of the interdependent sets of Π terms is meaningful in describing the same physical phenomenon. Furthermore, one can obtain all interdependent sets by combinations of products of powers arising from only one of these interdependent sets. Karam & Saad (2021) suggests that the utility in having all of these interdependent sets is that some may be more preferable given the specific purposes and measurements from experiments. It is in fact this notion of preferable Π terms, as well as the sets thereof, which drives the possibility of measuring that preferability via uncertainty propagation, but that will be focused on in a later section.

3.3.1 Mathematical Procedure Through an Example

The method of repeating variables is described by Karam & Saad (2021) and Zohuri (2015), and has already been implemented by the former in Python (which their paper discusses). The method follows the steps outlined below, as for the example with the pendulum which was introduced before. Guidance for this specific example was also obtained from Rosa (2021).

1. Begin with the dimensional matrix for the physical phenomenon (which is the phenomenon

of the swinging pendulum in this case),

$$\boldsymbol{M} = \begin{matrix} \\ \\ \\ l \end{matrix} \begin{matrix} \begin{pmatrix} 0 & -2 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \end{pmatrix} \\ g \quad m \quad \tau \quad \theta_0 \end{matrix} \begin{matrix} T \\ M \\ L \end{matrix} \tag{5}$$

where each row pertains to a dimension, and each column pertains to each physical variable. The ordering of the rows is arbitrary, however the ordering of the columns must ensure that the dimensional matrix is non-singular if a solution is sought.

There are notable characteristics of this dimensional matrix that must be noted however, which would best be demonstrated through this next equation. Say that $\boldsymbol{M}$ is included as before, but in the matrix equation

$$\begin{matrix} \begin{pmatrix} 0 & -2 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \end{pmatrix} \\ l \quad g \quad m \quad \tau \quad \theta_0 \end{matrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} -2x_2 - x_4 \\ x_3 \\ x_1 + x_2 \end{pmatrix} \tag{6}$$

where the physical variables are denoted below each column of the dimensional matrix. Here, the dimensional matrix actually gives a relationship between the powers of the physical variables within a monomial, and the powers of the resulting fundamental dimensions. In other words, the dimensional matrix outlines the mapping of the monomial $l^{x_1} g^{x_2} m^{x_3} \tau^{x_4} \theta^{x_5}$ to its resulting dimension, $T^{(-2x_2 - x_4)} M^{(x_3)} L^{(x_1 + x_2)}$. Therefore, if we are looking for the powers of these physical variables as a monomial which yields a dimension of $T^0 M^0 L^0$ (the powers within a Buckingham Π term), then we would wish to solve

$$\begin{pmatrix} 0 & -2 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}. \tag{7}$$

The particular vector containing $x_1 \dots x_5$ which we are solving for is in fact called the kernel vector of the (dimensional) matrix, $\boldsymbol{M}$. These kernel vectors are the vectors which yield a zero vector when multiplied with the dimensional matrix. Such kernel vectors therefore span what is called the null space of a matrix (also said to be a basis of the null space when they are linearly independent from other kernel vectors).

The next characteristic of the dimensional matrix which is key is its nullity. This is the number of vectors which constitute a basis for the null space. In other words it signifies the number of dimensions the null space has. The nullity will therefore give the number of non-trivial kernel vectors, which in turn will allow us to construct all the non-trivial Buckingham Π terms. The rank-nullity theorem provides the means to determine the nullity (Contreras et al., 2018), where for a matrix $\boldsymbol{A}$, this is $\dim(\mathrm{N}(\boldsymbol{A})) = n - r$, where n is the number of columns of $\boldsymbol{A}$, and r is the rank of $\boldsymbol{A}$.

With the example of the pendulum then, the number of rows of $\boldsymbol{M}$ is 5, and the rank of the matrix is 3. Therefore, $n = 5$ and $r = 3$. This means that the nullity, and so the number of null space basis (kernel) vectors, is 2. The rank of a matrix is the same as the number of linearly independent columns (the number of dimensions the column space of the matrix has), as well as the number of linearly independent physical variables according

to their dimensions. And so, what the nullity signifies is the number of physical variables whose dimensions are linearly dependent. This follows through logically if we remember that with only linearly independent physical variables, no non-trivial Buckingham Π term can be found since each variable has a dimension which cannot possibly be cancelled out by another variable's dimensions.

2. Next, the dimensional matrix is split, separating the linearly independent "repeating" variables and the linearly dependent "non-repeating" variables into two sub-matrices. The reason that these groups of variables are either repeating or non-repeating will become clear in a few steps. And so, the original dimensional matrix in our example becomes split such as in

$$\boldsymbol{M} = \left(\begin{array}{ccc|cc} 0 & -2 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \end{array} \right) \qquad \begin{matrix} \uparrow \\ \boldsymbol{P} \end{matrix} \quad \begin{matrix} \uparrow \\ \boldsymbol{Q} \end{matrix} \tag{8}$$

where $\boldsymbol{P}$ is the matrix representing all linearly independent repeating variables, and $\boldsymbol{Q}$ is the matrix representing all linearly dependent non-repeating variables. $\boldsymbol{P}$ is always a square matrix of the same number of rows as the dimensional matrix (which has a full rank, of value 3). It is important that $\boldsymbol{P}$ is not singular. Otherwise, no combination of the repeating variables (in $\boldsymbol{P}$) would be able to yield a dimensionless Π term when combined with any single non-repeating variable (in $\boldsymbol{Q}$). If $\boldsymbol{P}$ is rank deficient (and thus singular), then no solution can be obtained. In such a case, the physical variable columns of the original dimensional matrix must be rearranged so that $\boldsymbol{P}$ is non-singular, which corresponds to $\boldsymbol{P}$ truly containing linearly independent repeating variables. This also depends on whether the original dimensional matrix is rank deficient or not. Hence, the original choice of fundamental base dimensions and variables must be appropriate so that the dimensional matrix is not singular. The number of non-repeating variables is also the same as the number of expected Π terms (the number of dimensions of the null space of the dimensional matrix).

3. Once the dimensional matrix is formed, and it can be split into the matrices $\boldsymbol{P}$ and $\boldsymbol{Q}$ as in Equation 8, then Equation 7 can be manipulated to be easily solved. But first we must step back for a moment. We know that we need to find the null space basis kernel vectors of a given dimensional matrix, $\boldsymbol{M}$, and that each such vector can be found by solving

$$\underset{[m\times n]}{\boldsymbol{M}} \; \underset{[n\times 1]}{\boldsymbol{x}} = \underset{[m\times 1]}{\boldsymbol{0}} \tag{9}$$

for $\boldsymbol{x}$. For the specific example of the pendulum, this equation becomes

$$\underset{[3\times 5]}{\boldsymbol{M}} \; \underset{[5\times 1]}{\boldsymbol{x}} = \underset{[3\times 1]}{\boldsymbol{0}} \, . \tag{10}$$

Now, solving Equation 10 with only $\boldsymbol{M}$ and $\boldsymbol{0}$ being known is not possible since the system is underdetermined (3 equations and 5 unknowns). It is here that we use the fact that at least one non-repeating variable must be present in the Π term in order for the term to be non-trivial. First, the kernel vector must be split into the elements which get multiplied by the repeating variables (vector $\boldsymbol{v}_i$), and those which get multiplied by the non-repeating variables (vector $\boldsymbol{e}_i$), such that

$$\boldsymbol{x}_i = \begin{bmatrix} \boldsymbol{v}_i \\ \boldsymbol{e}_i \end{bmatrix} . \tag{11}$$

Since at least one non-repeating variable must be present in each Π term, each one can be set to have an exponent of one whilst all others are set to zero. As Zohuri (2015)

explains, having any other set values for $\boldsymbol{e}_i$ is not needed since all other cases would be linear combinations of the aforementioned pattern. Thus, the two kernel vectors for $\boldsymbol{M}$ would be

$$\boldsymbol{x}_1 = \begin{pmatrix} \boxed{a_1} \\ \boxed{b_1} \\ \boxed{c_1} \\ \boxed{1} \\ \boxed{0} \end{pmatrix} \begin{matrix} \\ \\ \\ \longleftarrow \boldsymbol{v}_1 \\ \longleftarrow \boldsymbol{e}_1 \end{matrix} \quad \text{and} \quad \boldsymbol{x}_2 = \begin{pmatrix} \boxed{a_2} \\ \boxed{b_2} \\ \boxed{c_2} \\ \boxed{0} \\ \boxed{1} \end{pmatrix} \begin{matrix} \longleftarrow \boldsymbol{v}_2 \\ \\ \\ \longleftarrow \boldsymbol{e}_2 \\ \\ \end{matrix}. \tag{12}$$

As such, vectors $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$ are canonical basis vectors, such that each is a column of the appropriately sized identity matrix (corresponding to the number of non-repeating variables).

Hence, given a single dimensional matrix, there exists one or more kernel vectors (and so Π terms also) due to the sequential setting of non-repeating variables. Within this set of Π terms, each non-repeating variable only appears once in a single Π term, and so those variables are termed non-repeating variables. However, all repeating variables get to be present in any number of Π terms within such a set, and so they are termed repeating variables.

4. All the relevant components are now present to find the unknowns within each kernel vector, which correspond to vectors $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$. Equation 9 can be fundamentally rewritten in order to solve for each unknown component of the kernel vectors. The resulting equation becomes

$$\begin{bmatrix} \boldsymbol{P} & \boldsymbol{Q} \end{bmatrix} \begin{bmatrix} \boldsymbol{v}_i \\ \boldsymbol{e}_i \end{bmatrix} = \boldsymbol{0} \quad \Rightarrow \quad \boldsymbol{P}\boldsymbol{v}_i + \boldsymbol{Q}\boldsymbol{e}_i = \boldsymbol{0} \quad \Rightarrow \quad \boldsymbol{v}_i = -\boldsymbol{P}^{-1}\boldsymbol{Q}\boldsymbol{e}_i. \tag{13}$$

And so, all the pieces are put together. Everything on the right hand side of the equation $\boldsymbol{v}_i = -\boldsymbol{P}^{-1}\boldsymbol{Q}\boldsymbol{e}_i$ is known. All that needs to be done are the actual matrix calculations for each vector $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$. Thus, for $\boldsymbol{v}_1$ we have

$$\boldsymbol{v}_1 = -\begin{pmatrix} 0 & -2 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}^{-1} \begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1/2 \\ -1/2 \\ 0 \end{pmatrix}, \tag{14}$$

and for $\boldsymbol{v}_2$ we have

$$\boldsymbol{v}_2 = -\begin{pmatrix} 0 & -2 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}^{-1} \begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}. \tag{15}$$

Therefore

$$\boldsymbol{x}_1 = \begin{pmatrix} 1/2 \\ -1/2 \\ 0 \\ 1 \\ 0 \end{pmatrix} \quad \text{and} \quad \boldsymbol{x}_2 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}. \tag{16}$$

Since the kernel vectors have been found, then the Π terms for this specific set can be constructed. The way to do this is to use the original order of the columns in the dimensional matrix $\boldsymbol{M}$. Thus, the elements of $\boldsymbol{x}_1$ correspond to the powers of l, g, m, τ, and θ_0 respectively. And so, finally we have

$$\Pi_1 = \sqrt{\frac{l}{g}}\tau \quad \text{and} \quad \Pi_2 = \theta_0. \tag{17}$$

Note that τ and θ_0 are the non-repeating variables, and that all others are the repeating variables.

5. The only thing remaining is to repeat the process (steps 1 to 4) but each time changing the order of the columns in the original dimensional matrix, so that several sets of the same number of Π terms can be found. For example, the dimensional matrix

$$\boldsymbol{M'} = \begin{matrix} \begin{pmatrix} -1 & 0 & 0 & -2 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix} \begin{matrix} T \\ M \\ L \end{matrix} \\ \tau \quad m \quad l \quad g \quad \theta_0 \end{matrix} \tag{18}$$

would yield the kernel vectors

$$\boldsymbol{x}'_1 = \begin{pmatrix} -2 \\ 0 \\ -1 \\ 1 \\ 0 \end{pmatrix} \qquad \text{and} \qquad \boldsymbol{x}'_2 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}. \tag{19}$$

and the Π terms

$$\Pi'_1 = \frac{g}{\tau^2 l} \qquad \text{and} \qquad \Pi'_2 = \theta_0. \tag{20}$$

For this second iteration as it were, the non-repeating variables are this time g and θ_0. This yields a non-duplicate Π term compared to the previous Π_1 and Π_2 terms, but one that is still a transformation of Π_1 (and hence is interdependent as mentioned before). Then more and more sets Π terms are gathered in this manner. For this problem in particular, it can be seen that all dimensional matrices which have θ_0 as a repeating variable will be rank deficient, and there would be other possible configurations of the dimensional matrix which also are rank deficient.

The total number of maximum sets of Π terms is the same as all the possible permutations of the dimensional matrix that allow each unique group of non-repeating variables to occur once. However, some of these combinations may be a duplicate of another set of Π terms (where just the corresponding Π terms are just in a different order compared to another set), or simply may be rank deficient. Since the number of non-repeating variables is the same as the number of Π terms in a single set that are expected (if the dimensional matrix is of a full rank), this maximum number can be determined by

$$n_{\text{perms}} = C(n_{\text{v}}, n_{\text{nrv}}) = C(n_{\text{v}}, n_{\Pi}) \tag{21}$$

where n_{perms} is the number of permutations, n_{v} is the number of total physical variables, n_{nrv} is the number of non-repeating variables, and n_{Π} is the number of Π terms (Karam & Saad, 2021).

3.4 Implementation in Haskell

3.4.1 Structure

Before the overall structure of the Haskell implementation is explained, first the data structures that were used must be explained. Listing 1 shows the custom algebraic data type definitions that provide the basis for defininining all further functions.

```
1  data Dim = A | B | C | D | E | F | G | H | I | J | K | L | M | N | O
     | P | Q | R | S | T | U | V | W | X | Y | Z
2  deriving (Eq, Show, Ord, Read)
3
4  type Pow = Rational
5  type Var = String
6  type DimQu = [(Dim,Pow)]
7  type DimMat = DM.Matrix Pow
8  type Kernel = DM.Matrix Pow
```

Listing 1: Custom algebraic data types for method of repeating variables.

Line one in Listing 1 shows the creation of a new data type `Dim`, with the values ranging from `A` to `Z`. These are used to represent the Dimensions of a quantity, such as "`L`" for length, or "`T`" for time. Lines 4 to 8 are all type synonym declarations to increase the clarity and conciseness of the code. `Pow` on line 4 is made to be equivalent to the `Rational` data type. An example of a `Rational` (or `Pow`) value would be "`1 % 3`" which represents 1/3. `Var` on Line 5 is made equivalent to the `String` data type. This type is used to represent the physical variable names, such as "`distance`" or "`\theta_0`".

`DimQu` on line 6 is made equivalent to a list of tuples, each containing a dimension, and a power of that dimension (to form a dimensional quantity). For example, speed would have a dimensional quantity of `[(L,1 % 1),(T,-1 % 1)]`. `DimMat` and `Kernel` on lines 7 and 8 are made equivalent to the type `Matrix Pow`, which represents matrices that have elements of type `Pow` (made equivalent to `Rational`). It is worth noting here that the main external libraries used were the `Data.Matrix` and `Numeric.Matrix` libraries, both imported as `DM` and `NM` respectively (the former of which can be seen on lines 7 and 8).

With this information, Figure 3.1 can be seen to show the structure of the Haskell implementation used to solve this problem of finding the Buckingham Π terms from a list of physical variables with their dimensions stated. The flow chart in Figure 3.1 specifically shows the ordering of the functions used, and the type signatures of each function. As such, just to be clear, the arrows do not represent functions, but simply show where the output of the previous function gets passed on to. The first and last functions in Figure 3.1 are simply textual parsing functions that allow the input to be text, and the output to be the Π terms formatted in a LaTeX equation format. However, the five functions in the middle are the primary data processing parts of the Haskell program.

It is worth noting that there are many smaller functions that have been used, but whose definitions are not included in this paper for the sake of conciseness. The names of these functions should be self explanatory (for example, `getRank` which returns the rank of a given matrix), but should any more clarification be needed, the entire source code can be found on https://github.com (see Symonds Patel (2023)).

3.4.2 Primary Functions

In order to dissect what the primary functions do, it is probably best again to work with the pendulum example from Section 3.3.1. First we can begin with the textual input as shown in Listing 2. This input represents each physical variable along with the dimensions which each holds. This whole input string including the newline characters (which cannot be seen) is assigned to `inputString` in Listing 2.

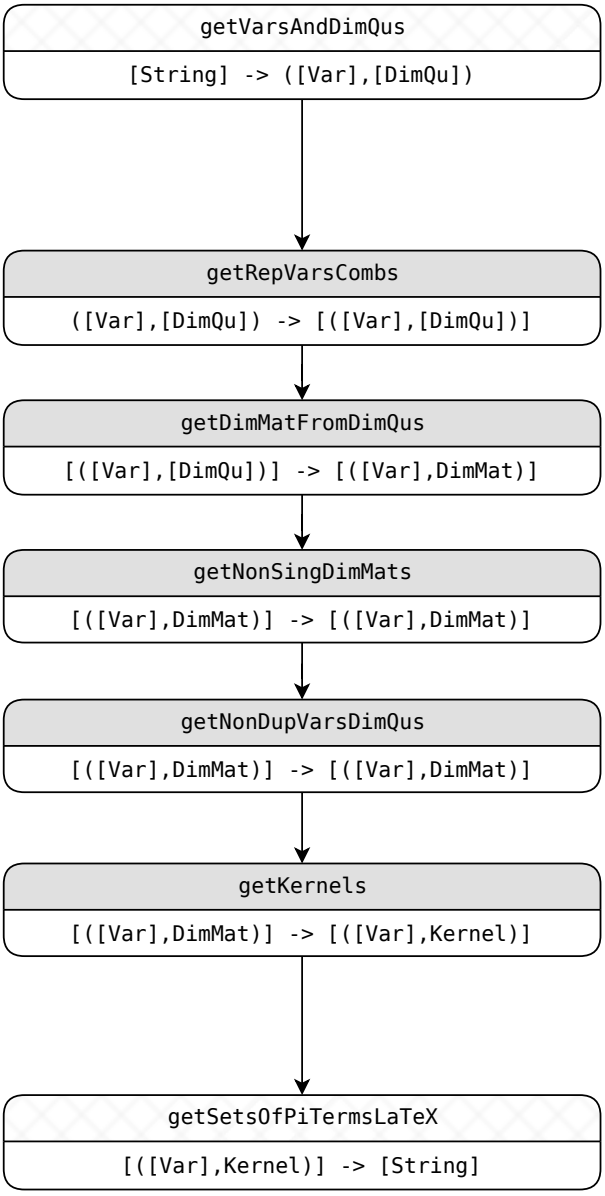

Figure 3.1: Flow chart showing the structure of the method of repeating variables implemented in Haskell.

```
1 inputString = "l = L
2                m = M
3                g = LT^{-2}
4                \theta_0 = L^{0}
5                \tau = T"
```

Listing 2: Textual input segment for the pendulum example.

The result shown in Listing 3 is of the type `([Var],[DimQu])`, and is the output of the function `getVarsAndDimQus`, which converts textual input into the data types defined in Listing 1, so that it can be processed.

```
GHCi> getVarsAndDimQus $ lines inputString
(["l ","m ","g ","\theta_0 ","\tau "],
[[(L,1 \% 1),(M,0 \% 1),(T,0 \% 1)],
 [(L,0 \% 1),(M,1 \% 1),(T,0 \% 1)],
 [(L,1 \% 1),(M,0 \% 1),(T,(-2) \% 1)],
 [(L,0 \% 1),(M,0 \% 1),(T,0 \% 1)],
 [(L,0 \% 1),(M,0 \% 1),(T,1 \% 1)]])
```

Listing 3: Output from `getVarsAndDimQus` with the pendulum example.

The function `lines` in Listing 3 simply converts a string that has newline characters, into a list of strings cut off by those newline characters (`lines :: String -> [String]`).

Once the input is parsed through `getVarsAndDimQus`, it can then be processed by the function `getRepVarsCombs` (Listing 4), which obtains all the different orderings of the list of dimensional quantities (`[DimQu]`) from which the dimensional matrices are built (where all the columns in each are built according to the order to the list of dimensional quantities). This way, all the combinations of non-repeating variables can be made to begin with, from which the respective dimensional matrices can be built. The list of variable names (`[Vars]`) for each list of dimensional quantities also has its order shifted in the same manner, so that each variable name still lines up with the same dimensional quantity.

```
getRepVarsCombs :: ([Var],[DimQu]) -> [([Var],[DimQu])]
getRepVarsCombs (vars,dimQus) = zip varsCombs dimQusCombs
  where nNonRepVars = getNPiTerms . getLUDecompU . makeDimMat $ dimQus
        dimQusCombs = getRepVarsCombsPM nNonRepVars dimQus :: [[DimQu]]
        varsCombs = getRepVarsCombsPM nNonRepVars vars :: [[Var]]
```

Listing 4: Definition of `getRepVarsCombs`.

It would be too long to show the output of `getRepVarsCombs`, but the length of the list of different combinations can be seen in Listing 5.

```
GHCi> length $ getRepVarsCombs . getVarsAndDimQus $ lines inputString
10
```

Listing 5: length of the output from `getRepVarsCombs` with the pendulum example.

The length shown in Listing 5 can be seen to be the same as $C(5,2)$, as in Equation 21.

Next, each dimensional matrix is built from each list of dimensional quantities (`[DimQu]`) with `getDimMatFromDimQus` (Listing 6).

```
getDimMatFromDimQus :: [([Var],[DimQu])] -> [([Var],DimMat)]
getDimMatFromDimQus varsDimQusList = (map . fmap) makeDimMat
   varsDimQusList
```

Listing 6: Definition of `getDimMatFromDimQus`.

Again, it would be too long to give the whole list of 10 tuples containing the variable names with the corresponding dimensional matrix of each (`([Var],DimMat)`), but the first element of the list has been given for clarity in Listing 7.

```
GHCi> head $ getDimMatFromDimQus . getRepVarsCombs . getVarsAndDimQus $
    lines inputString
(["g ","\theta_0 ","\tau ","l ","m "],
|                                                   |
|    1 \% 1    0 \% 1    0 \% 1    1 \% 1    0 \% 1 |
|    0 \% 1    0 \% 1    0 \% 1    0 \% 1    1 \% 1 |
| (-2) \% 1    0 \% 1    1 \% 1    0 \% 1    0 \% 1 |
|                                                   |)
```

Listing 7: The first element of the output from `getDimMatFromDimQus` with the pendulum example.

Now that the list of all the possible dimensional matrices which provide all combinations of non-repeating variables have been made, all of the matrices which are singular need to be filtered out. This is what `getNonSingDimMats` does (Listing 8), by taking out any tuple containing a dimensional matrix which has a determinant of zero from the list it was given (`:: [([Var],DimMat)]`).

```
getNonSingDimMats :: [([Var],DimMat)] -> [([Var],DimMat)]
getNonSingDimMats varsDimMatList = filter filterFunc varsDimMatList
  where filterFunc = \(_,dimMat) -> if (getDet dimMat /= 0) then True
   else False
```

Listing 8: Definition of `getNonSingDimMats`.

Listing 9 shows the output of `getNonSingDimMats`, where all but three matrices are found to be non-singular.

```
GHCi> getNonSingDimMats . getDimMatFromDimQus . getRepVarsCombs .
   getVarsAndDimQus $ lines inputString
[(["m ","g ","\tau ","l ","\theta_0 "],
|                                                   |
|    0 \% 1    1 \% 1    0 \% 1    1 \% 1    0 \% 1 |
|    1 \% 1    0 \% 1    0 \% 1    0 \% 1    0 \% 1 |
|    0 \% 1 (-2) \% 1    1 \% 1    0 \% 1    0 \% 1 |
|                                                   |),
(["l ","m ","\tau ","g ","\theta_0 "],
|                                                   |
|    1 \% 1    0 \% 1    0 \% 1    1 \% 1    0 \% 1 |
|    0 \% 1    1 \% 1    0 \% 1    0 \% 1    0 \% 1 |
|    0 \% 1    0 \% 1    1 \% 1 (-2) \% 1    0 \% 1 |
|                                                   |),
(["l ","m ","g ","\theta_0 ","\tau "],
|                                                   |
|    1 \% 1    0 \% 1    1 \% 1    0 \% 1    0 \% 1 |
|    0 \% 1    1 \% 1    0 \% 1    0 \% 1    0 \% 1 |
|    0 \% 1    0 \% 1 (-2) \% 1    0 \% 1    1 \% 1 |
|                                                   |)]
```

Listing 9: The output from `getNonSingDimMats` with the pendulum example.

Then, `getNonDupVarsDimQus` is used to further filter out any extra tuples containing dimensional matrices that will give equivalent Π terms (Listing 10). `getNonDupVarsDimQus` does this by seeing if a dimensional matrix has a reduced echelon form equivalent to any other dimensional matrix. This is because the null space of a matrix is equivalent to the null space of the reduced row echelon form of the same matrix (Strang, 2006). Hence, any two dimensional matrices with the same reduced row echelon form, will have the same null space, and therefore the same kernel

vectors (and Π terms).

Applying `getNonDupVarsDimQus` to the output of `getNonSingDimMats` does not filter out any subsequent tuples, and so the output of `getNonDupVarsDimQus` is the same as in Listing 9.

```
getNonDupVarsDimQus :: [([Var],DimMat)] -> [([Var],DimMat)]
getNonDupVarsDimQus varsDimMatList = nonDupVarsRREFDimMatList
  where rrefDimMats = map getRREF . snd . unzip $ varsDimMatList
        varsDimMatRREFList = (map . fmap) ((fromMaybe (DM.zero 1 1)) .
   getRREF) varsDimMatList
        nonDupRREFDimMats = map (fromMaybe (DM.zero 1 1)) . filter
   isJust $ nub rrefDimMats
        nonDupVarsRREFDimMatList =
          [fromMaybe ([""],DM.zero 1 1) . find ((== nonDupRREFDimMat) .
    ((fromMaybe (DM.zero 1 1)) . getRREF) . snd) $ varsDimMatList |
   nonDupRREFDimMat <- nonDupRREFDimMats]
```

Listing 10: Definition of `getNonDupVarsDimQus`.

Then, the kernel vectors of each dimensional matrix is found via `getKernels` (Listing 11). The same equation as Equation 13 is used, however instead of using the $\boldsymbol{v}_i = -\boldsymbol{P}^{-1}\boldsymbol{Q}\boldsymbol{e}_i$ to find each unknown segment of each kernel vector, `getKernels` uses $\boldsymbol{v} = -\boldsymbol{P}^{-1}\boldsymbol{Q}\boldsymbol{e}$, where $\boldsymbol{e}$ is the canonical basis matrix (as opposed to a canonical basis vector, $\boldsymbol{e}_i$). Thus, for this example, instead of individual kernel vectors, we can obtain a kernel matrix,

$$\boldsymbol{x} = \begin{pmatrix} a_1 & a_2 \\ b_1 & b_2 \\ c_1 & c_2 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}, \tag{22}$$

where a_1 to c_2 can all be found via one evaluation of the equation $\boldsymbol{v} = -\boldsymbol{P}^{-1}\boldsymbol{Q}\boldsymbol{e}$, where

$$\boldsymbol{e} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}. \tag{23}$$

That way, the unknowns of each kernel vector can be obtained using one computation instead of gluing together the results of several computations.

```
getKernels :: [([Var],DimMat)] -> [([Var],Kernel)]
getKernels varsDimMatList = (map . fmap) getKernel varsDimMatList
  where getKernel dimMat = (DM.scaleMatrix (-1) (DM.multStd xInv (DM.
   multStd y canVecBasMat))) DM.<-> canVecBasMat
        where dimMatRREF = fromMaybe (DM.zero 1 1) $ getRREF dimMat
              (x,y) = getXY dimMatRREF
              xInv = fromMaybe (DM.zero 1 1) $ getInverse x
              canVecBasMat = DM.identity $ getNPiTerms dimMatRREF
```

Listing 11: Definition of `getKernels`.

Listing 12 shows the output of `getKernels` for the pendulum example, where each kernel matrix has been found for each dimensional matrix. What we are left with is a list of tuples, each containing a list of variable names, and a kernel matrix, where each column thereof is a single Π term (and each row corresponds to the respective variable according to the list of variable names). Thus, each kernel matrix contains a set of Π terms.

```
GHCi> getKernels . getNonDupVarsDimQus . getNonSingDimMats .
   getDimMatFromDimQus . getRepVarsCombs . getVarsAndDimQus $ lines
   inputString
[(["m ","g ","\tau ","l ","\theta_0 "],
|                       |
|     0 \% 1     0 \% 1 |
| (-1) \% 1     0 \% 1 |
| (-2) \% 1     0 \% 1 |
|     1 \% 1     0 \% 1 |
|     0 \% 1     1 \% 1 |
|                       |),
(["l ","m ","\tau ","g ","\theta_0 "],
|                       |
| (-1) \% 1     0 \% 1 |
|     0 \% 1     0 \% 1 |
|     2 \% 1     0 \% 1 |
|     1 \% 1     0 \% 1 |
|     0 \% 1     1 \% 1 |
|                       |),
(["l ","m ","g ","\theta_0 ","\tau "],
|                       |
|     0 \% 1 (-1) \% 2 |
|     0 \% 1     0 \% 1 |
|     0 \% 1     1 \% 2 |
|     1 \% 1     0 \% 1 |
|     0 \% 1     1 \% 1 |
|                       |)]
```

Listing 12: The output from `getKernels` with the pendulum example.

This whole process can be wrapped into a single function via function composition so that we have `getSetsOfPiTerms` (Listing 13).

```
getSetsOfPiTerms :: ([Var],[DimQu]) -> [([Var],Kernel)]
getSetsOfPiTerms = getKernels .
                   getNonDupVarsDimQus .
                   getNonSingDimMats .
                   getDimMatFromDimQus .
                   getRepVarsCombs
```

Listing 13: Definition of `getSetsOfPiTerms`.

And so, all that is left to do is to parse the result from Listing 12 to text, which is what `getSetsOfPiTermsLaTeX` does, and in a format that is comprehensible by LaTeX. Thus, to finish off our example, we can obtain the final output as a LaTeX equation in Equation 24, where each row corresponds to a single set of Π terms.

$$\begin{aligned} \Pi_1 &= g^{-1}\tau^{-2}l & \Pi_2 &= \theta_0 \\ \Pi_1 &= l^{-1}\tau^{2}g & \Pi_2 &= \theta_0 \\ \Pi_1 &= \theta_0 & \Pi_2 &= l^{-1/2}g^{1/2}\tau \end{aligned} \tag{24}$$

Hopefully by running through the main components of the Haskell implementation via an example, the structure of the whole implementation is mostly clear. It is most likely that the smaller algorithms and functions used within each of the primary functions are not as clear, however the aim of this section was only to explain the skeletal structure of the implementation. This is because the key features of using Haskell show themselves significantly at the higher levels of

the implementation rather than the lower levels. It is precisely the smaller algorithms in this Haskell implementation that are likely to be insignificant (although possibly done in a novel way compared to typically imperative languages) because they do not pull largely from Haskell's pot of unique features.

4 Uncertainty Propagation Through Buckingham Pi Terms

4.1 Motivation

The previous section dealt with obtaining sets of Π terms via the method of repeating variables. Now, because there may be a large selection of resulting Π terms, the next natural step in using dimensional analysis here is to pick those Π terms, or sets of Π terms according to which is best suited for the real experiment.

Karam & Saad (2021) show an example of why selecting the appropriate Π terms is important. The premise of their example is essentially that because variables may be non-repeating variables, they may not always appear in some sets. Therefore, only certain sets of Π terms may qualify in containing the variables of interest, in the way that is needed to express an experiment as an appropriate functional relationship.

Now, even then, there may be some sets of Π terms which are equally desirable on the surface, as well as individual Π terms which seem equally viable to use. Thus, another metric would be needed to decide which Π term (or set thereof) is best to use in order to set up the experiment and retrieve meaningful data.

This section details the method (with its implementation) that is able to build that metric, which is founded on the natural variability associated which each physical variable. In other words, since physical variables may not simply have a single value associated with them, but a distribution of values (a probability distribution), then so too the resulting Π term would have an associated probability distribution. Therefore, the physical variables and Π terms are treated as random variables instead of standard variables.

Let us use the example from the previous section, and say that we took interest in the Π term

$$\Pi_2 = l^{-1/2} g^{1/2} \tau, \tag{25}$$

found in the third row of Equation 24. Now, if we realised that in fact l was actually a random variable with a known probability distribution, X_1, and so was g (X_2) and τ (X_3), then we would be prompted to find out what the probability distribution of the Π term would be as a whole. Hence, we would want to find out the nature of the probability distribution of Y in the corresponding formulation

$$Y = X_1^{-1/2} X_2^{1/2} X_3, \tag{26}$$

where Y is a function of the three original known random variables, corresponding to the same monomial as the original Π term in Equation 25.

It is the final aim of this paper to provide a means to obtain those exact probability distributions associated with Π terms, from the distributions that may be associated to physical variables within those Π terms. It is not one of the aims of this paper however, to decide which of the resultant Π terms (represented as their probability distributions) are best. This is up to the discretion of those carrying out their respective experiments, since each experiment or study would naturally have different frameworks as to what constitutes the best.

4.2 Mathematical Procedure

4.2.1 A Function of Several Random Variables

The key principle behind this section is that several known random variables are being transformed by a function of those variables. Firstly, it is assumed that the known random variables associated with any physical variables are represented in the form of a joint probability density function (PDF). This way, any relationships between the random variables can be contained within such a representation. Hence, independence is not assumed between any of the physical variables.

Since this is taken to be the starting point, there is some amount of pre-processing needed in order to first have the joint PDF of the random variables associated to the relevant physical variables.

Lu (2018) provides the key equation for obtaining the cumulative distribution function (CDF) of the random variable Y, which is a transformation of the known random variables $X_1, X_2, \ldots, X_n$ (in other words, where $Y = g(X_1, X_2, \ldots, X_n)$), from the joint PDF of the known random variables. This equation is outlined as

$$F_Y(y) = P(\boldsymbol{X} \in R_x) = \int_{\boldsymbol{x} \in R_x} \cdots \int f_{\boldsymbol{X}}(x_1, x_2, \ldots, x_n)\, \mathrm{d}x_1 \ldots \mathrm{d}x_n \tag{27}$$

where $f_{\boldsymbol{X}}(x_1, x_2, \ldots, x_n)$ is the PDF of the known random variables, $F_Y(y)$ is the CDF of the transformation of the known random variables, the event $\{Y \leq y\} \equiv \{R_x | \boldsymbol{x} \in R_x, g(\boldsymbol{x}) \leq y\}$,

$$\boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \quad \text{and} \quad \boldsymbol{X} = \begin{pmatrix} X_1 \\ X_2 \\ \vdots \\ X_n \end{pmatrix}. \tag{28}$$

In other words, R_x is the set of all $\boldsymbol{x}$ values which satisfy the condition $g(\boldsymbol{x}) \leq y$. And so, R_x corresponds directly to the set of y values which satisfy $Y \leq y$, where the associated probability therefrom as an event is the definition of the CDF of Y, $F_Y(y)$. The set of $\boldsymbol{x}$ values within R_x therefore denotes also the region of integration of the joint PDF, $f_{\boldsymbol{X}}(x_1, x_2, \ldots, x_n)$, from which is gained a value for $F_Y(y)$.

Finally, by definition, the PDF of Y can be obtained as

$$f_Y(y) = \frac{\mathrm{d}}{\mathrm{d}y} \left(F_Y(y) \right). \tag{29}$$

Since this process may seem rather mathematically complex, in the subsequent parts, a few analytical examples have been done to give more clarity on the mathematical procedure.

Before those examples are seen however, it is worth noting here that the notion of getting $f_Y(y)$ from $f_{\boldsymbol{X}}(\boldsymbol{x})$ where Y is a transformation of $\boldsymbol{X}$ ($Y = g(\boldsymbol{X})$), is formally called a pushforward distribution (Gorelli, 2021). It is so termed because (in a simplified sense) we are "pushing" the inputs of $f_{\boldsymbol{X}}$ along the transformation g to get f_Y. This idea is at its core a formal formulation in category theory, but is the fundamental representation of the above problem. This is worth noting because this representation was unable to be used in the later shown Haskell implementation, but would be a more accurate representation.

4.2.2 Example One

For this first example we begin by wanting to find the PDF of the random variable Y, from the known random variables X_1 and X_2 (assumed to be independent), all as described by

$$\begin{aligned} X_1 &\sim \mathcal{U}(0,1), \\ X_2 &\sim \mathcal{U}(0,1), \\ Y &= g(X_1, X_2) = X_1 X_2. \end{aligned} \tag{30}$$

Since X_1 and X_2 are independent and uniformly distributed random variables between 0 and 1, we can derive the joint PDF of X_1 and X_2, $f_{X_1,X_2}(x_1, x_2)$, as being

$$\begin{aligned} & f_{X_1,X_2}(x_1, x_2) = f_{X_1}(x_1) f_{X_2}(x_2) \\ \implies & f_{X_1,X_2}(x_1, x_2) = \begin{cases} 1, & \text{if } (0 \leq x_1 \leq 1) \text{ and } (0 \leq x_2 \leq 1) \\ 0, & \text{otherwise.} \end{cases} \end{aligned} \tag{31}$$

This joint PDF can be seen plotted in Figure 4.1.

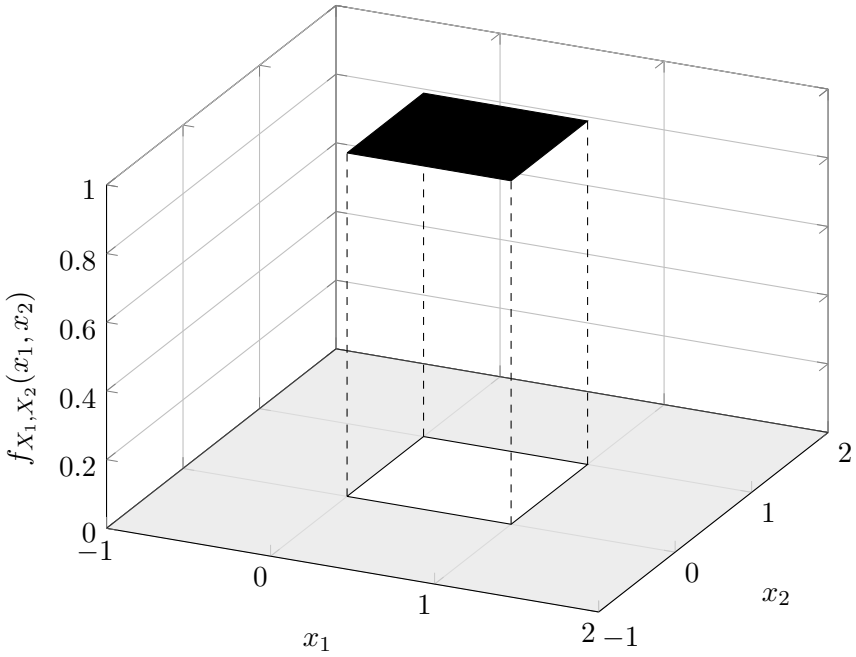

Figure 4.1: A plot showing the known joint PDF, $f_{X_1,X_2}(x_1, x_2)$, for Example One.

Using Equation 27, the CDF of Y can be written as

$$F_Y(y) = \iint\limits_{R=\{x_1 x_2 \leq y\}} f_{X_1,X_2}(x_1, x_2)\, \mathrm{d}x_1 \mathrm{d}x_2. \tag{32}$$

To illustrate this double integral, we can plot a two-dimensional representation of the region of integration (Figure 4.2).

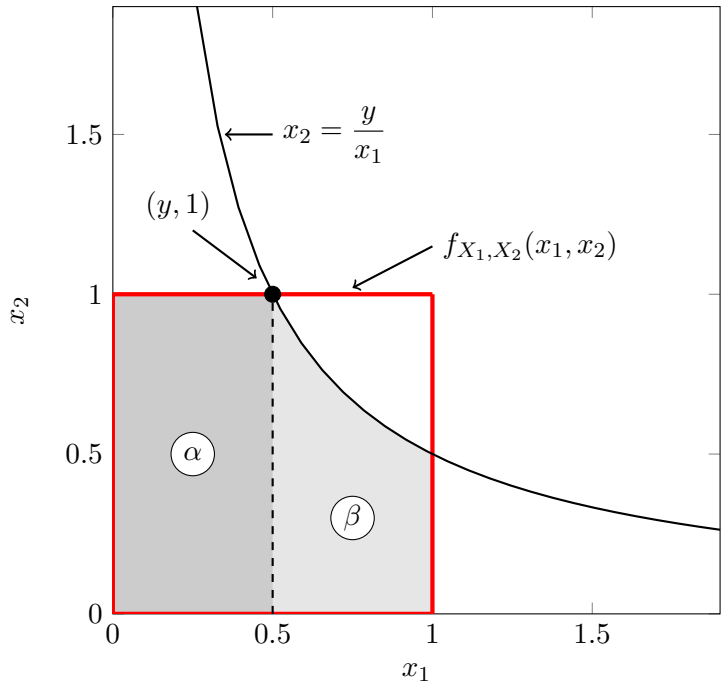

Figure 4.2: A plot showing the (split) region of integration that evaluates the CDF $F_y(y)$ in Example One.

In Figure 4.2, the region of integration is essentially the area which is underneath the curve $x_2 = \frac{y}{x_1}$, which represents the region $R = \{x_1 x_2 \leq y\}$. Since all values of $f_{X_1,X_2}(x_1, x_2)$ outside of $(0 \leq x_1 \leq 1)$ and $(0 \leq x_2 \leq 1)$ are 0, then the region of integration can further be narrowed to the areas α and β shown in Figure 4.2. Splitting the region of integration into α and β also simplifies the analytical evaluation of the integral. Therefore, the CDF of Y, $F_Y(y)$ can be written as

$$F_Y(y) = \int\int_{R=\{x_1 x_2 \leq y\}} f_{X_1,X_2}(x_1, x_2)\,\mathrm{d}x_1\mathrm{d}x_2 = (\alpha \times 1) + (\beta \times 1), \tag{33}$$

since the value of $f_{X_1,X_2}(x_1, x_2)$ above α and β is always 1. All that is then left to do in order to find $F_Y(y)$, is to evaluate α and β. For α, we have

$$\begin{aligned} \alpha &= \int_0^y \mathrm{d}x_1 \\ &= \Big[x_1\Big]_0^y \\ &= y, \end{aligned} \tag{34}$$

and for β, we have

$$\begin{aligned} \beta &= \int_y^1 \frac{y}{x_1}\,\mathrm{d}x_1 \\ &= \Big[y\ln(x_1)\Big]_y^1 \\ &= -y\ln(y). \end{aligned} \tag{35}$$

Therefore, we can find an expression for $F_Y(y)$ as being

$$\begin{aligned} & F_Y(y) = \alpha + \beta \\ \implies & F_Y(y) = y - y\ln(y). \end{aligned} \tag{36}$$

Since we no know $F_Y(y)$, we can then easily find the PDF of Y, $f_Y(y)$, as being

$$\begin{aligned} f_Y(y) &= \frac{\mathrm{d}}{\mathrm{d}y}\left(F_Y(y)\right) \\ \implies f_Y(y) &= -\ln(y). \end{aligned} \tag{37}$$

Finally, $f_Y(y)$ can be seen plotted in Figure 4.3.

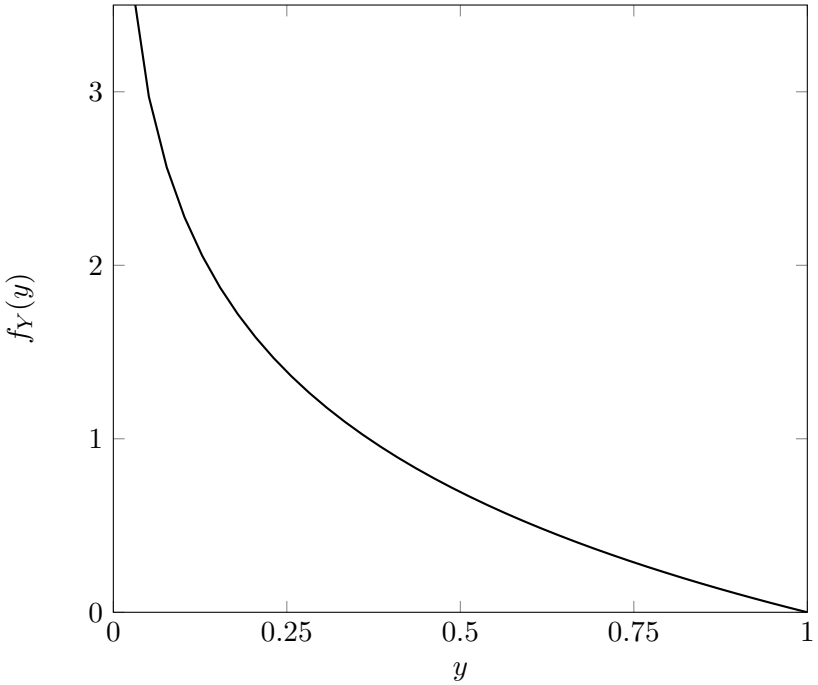

Figure 4.3: A plot showing the final PDF ,$f_Y(y)$, of the transformed random variable Y for Example One.

4.2.3 Example Two

For this example, we also begin by wanting to find the PDF of the random variable Y, from the known random variables X_1 and X_2 (assumed to be independent), both of which are described by

$$\begin{aligned} X_1 &\sim \mathcal{U}(0,1), \\ X_2 &\sim \mathcal{U}(0,1). \end{aligned} \tag{38}$$

However, this time the transformation $g(X_1, X_2)$ is different, and is such that

$$Y = g(X_1, X_2) = \frac{X_2}{X_1}. \tag{39}$$

As such, Equation 31 and Figure 4.1 still describe $f_{X_1,X_2}(x_1, x_2)$ for this example. Again, using Equation 27, we can write the CDF of Y, $F_Y(y)$, as

$$F_Y(y) = \int\int_{R=\{x_2/x_1 \leq y\}} f_{X_1,X_2}(x_1, x_2)\,\mathrm{d}x_1\mathrm{d}x_2, \tag{40}$$

where all is left unchanged besides the region of integration, R. This region of integration is shown in Figure 4.4 and Figure 4.5. In both plots, the region of integration is the area

underneath the curve $x_2 = yx_1$ (a rearrangement of $x_2/x_1 = y$), which is equivalent to the region $R = \{x_2/x_1 \leq y\}$. Since the value of $f_{X_1,X_2}(x_1, x_2)$ is again 0 outside of $(0 \leq x_1 \leq 1)$ and $(0 \leq x_2 \leq 1)$, the region of integration can be narrowed to being the areas α for when $0 \leq y \leq 1$, and β and γ for when $y > 1$. The reason that there are two descriptions/plots of the region of integration is that there is a discontinuity in the rate of change of area when $y = 1$ for the region under the curve $x_2 = yx_1$.

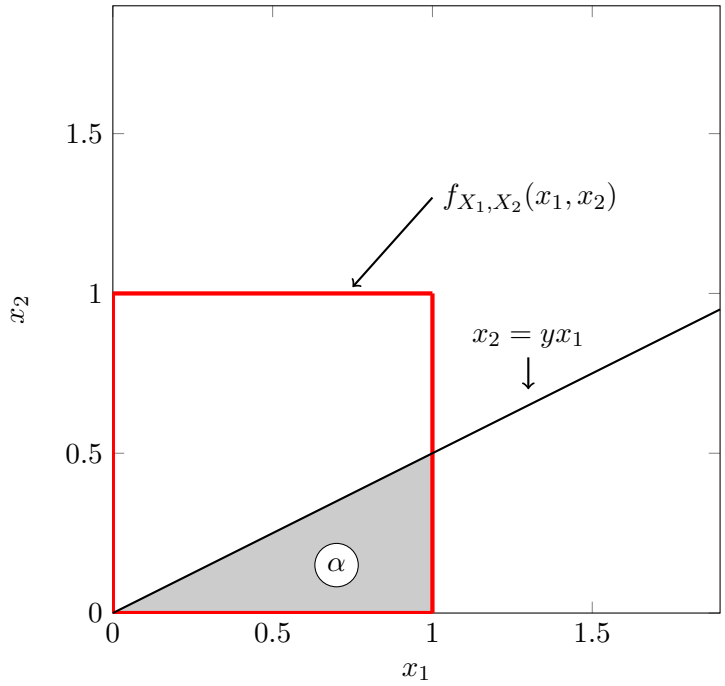

Figure 4.4: A plot showing the region of integration that evaluates the CDF $F_y(y)$ in Example Two, specifically for when $0 \leq y \leq 1$.

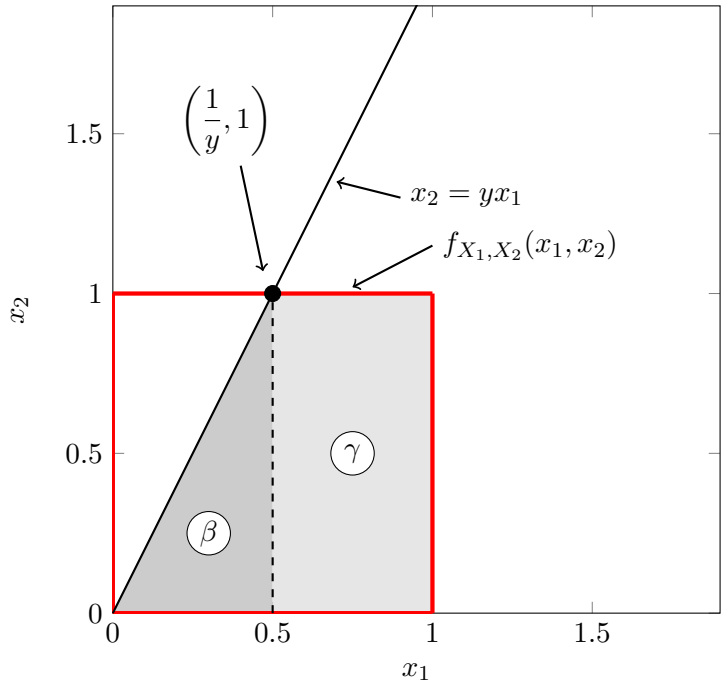

Figure 4.5: A plot showing the (split) region of integration that evaluates the CDF $F_y(y)$ in Example Two, specifically for when $y > 1$.

Thus, $F_Y(y)$ can be written as

$$F_Y(y) = \begin{cases} 0, & \text{if } y < 0 \\ \alpha, & \text{if } 0 \leq y \leq 1 \\ \beta + \gamma, & \text{if } y > 1, \end{cases} \tag{41}$$

since the value of $f_{X_1,X_2}(x_1, x_2)$ above the areas α, β, and γ is always 1. Thus, again, all that needs to be done to find $F_Y(y)$ is to evaluate the areas α, β, and γ. Firstly,

$$\begin{aligned} \alpha &= \int_0^1 yx_1 \, \mathrm{d}x_1 \\ &= \left[\frac{1}{2} y x_1^2 \right]_0^1 \\ &= \frac{1}{2} y. \end{aligned} \tag{42}$$

Secondly,

$$\begin{aligned} \beta &= \int_0^{\frac{1}{y}} yx_1 \, \mathrm{d}x_1 \\ &= \left[\frac{1}{2} y x_1^2 \right]_0^{\frac{1}{y}} \\ &= \frac{1}{2y}. \end{aligned} \tag{43}$$

Thirdly,

$$\begin{aligned} \gamma &= \int_{\frac{1}{y}}^1 \mathrm{d}x_1 \\ &= \left[x_1^2 \right]_{\frac{1}{y}}^0 \\ &= 1 - \frac{1}{y}. \end{aligned} \tag{44}$$

Thus, we have an expression for $F_Y(y)$ as being

$$F_Y(y) = \begin{cases} 0, & \text{if } y < 0 \\ \frac{1}{2} y, & \text{if } 0 \leq y \leq 1 \\ 1 - \frac{1}{2y}, & \text{if } y > 1. \end{cases} \tag{45}$$

Now, since we know $F_Y(y)$, we can easily find the PDF of Y, $f_Y(y)$, as being

$$\begin{aligned} f_Y(y) &= \frac{\mathrm{d}}{\mathrm{d}y} \left(F_Y(y) \right) \\ \implies f_Y(y) &= \begin{cases} 0, & \text{if } y < 0 \\ \frac{1}{2}, & \text{if } 0 \leq y \leq 1 \\ \frac{1}{2y^2}, & \text{if } y > 1. \end{cases} \end{aligned} \tag{46}$$

Finally $f_Y(y)$ for this example can be seen plotted in Figure 4.6.

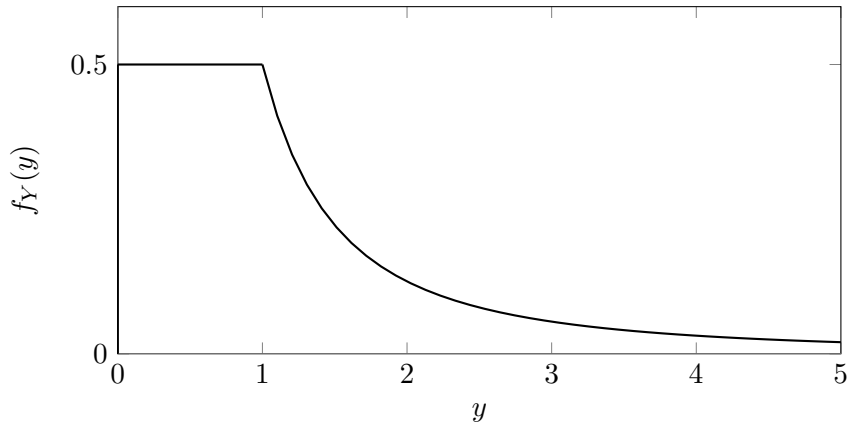

Figure 4.6: A plot showing the final PDF ,$f_Y(y)$, of the transformed random variable Y for Example Two.

4.2.4 Example Three

This example is separate from the previous two in that it involves different known random variables, in the manner in which they are distributed. That being said, we begin in the same way. X_1 and X_2 are known random variables, from which we wish to find the PDF of Y, $f_Y(y)$. Again, Y is a transformation of the known random variables X_1 and X_2 (assumed to be independent), and all of this is described as

$$\begin{aligned} X_1 &\sim \mathcal{N}(0,1), \\ X_2 &\sim \mathcal{N}(0,1), \\ Y = g(X_1, X_2) &= X_1 X_2. \end{aligned} \tag{47}$$

Notice that the transformation $g(X_1, X_2)$ is the same as in Example 1. Since both X_1 and X_2 are independent and are normally distributed (via the standard normal distribution), their joint PDF, $f_{X_1,X_2}(x_1, x_2)$, can be written as

$$\begin{aligned} f_{X_1,X_2}(x_1,x_2) &= f_{X_1}(x_1) f_{X_2}(x_2) \\ \implies f_{X_1,X_2}(x_1,x_2) &= \frac{1}{2\pi} e^{-\frac{1}{2}(x_1^2+x_2^2)}. \end{aligned} \tag{48}$$

The joint PDF is shown in Figure 4.7. Now, using Equation 27, the CDF of Y, $F_Y(y)$, can be written as

$$\begin{aligned} F_y(y) &= \iint\limits_{R=\{x_1 x_2 \leq y\}} f_{X_1,X_2}(x_1,x_2)\,\mathrm{d}x_1\mathrm{d}x_2 \\ &= \iint\limits_{R=\{x_1 x_2 \leq y\}} \frac{1}{2\pi} e^{-\frac{1}{2}(x_1^2+x_2^2)}\,\mathrm{d}x_1\mathrm{d}x_2. \end{aligned} \tag{49}$$

The region of integration, R, can be seen visualised in Figure 4.8, Figure 4.9, and Figure 4.10. Since $f_{X_1,X_2}(x_1, x_2)$ has non-zero values for all values of x_1 and x_2, we cannot narrow down the region of integration further like we have done before in Example 1 and Example 2. Figure 4.8, Figure 4.9, and Figure 4.10 show the region of integration, $R = \{x_1 x_2 \leq y\}$ for when $y = -1$, $y = 0$, and $y = 1$, respectively. They also show the nature of the region of integration more

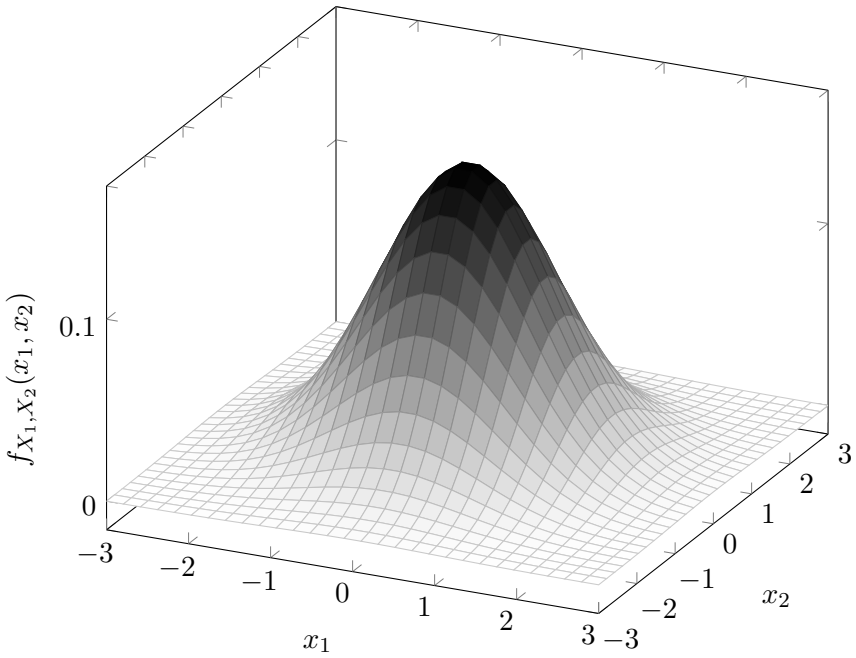

Figure 4.7: A plot showing the known joint PDF, $f_{X_1,X_2}(x_1, x_2)$, for Example Three.

generally for when $y < 0$ and $y \geq 0$. The process is exactly the same as it was in Example 1 specifically, however in this example, the domain needed to be evaluated is infinite in x_1 and x_2, and there is a wider range of y values to consider because of that.

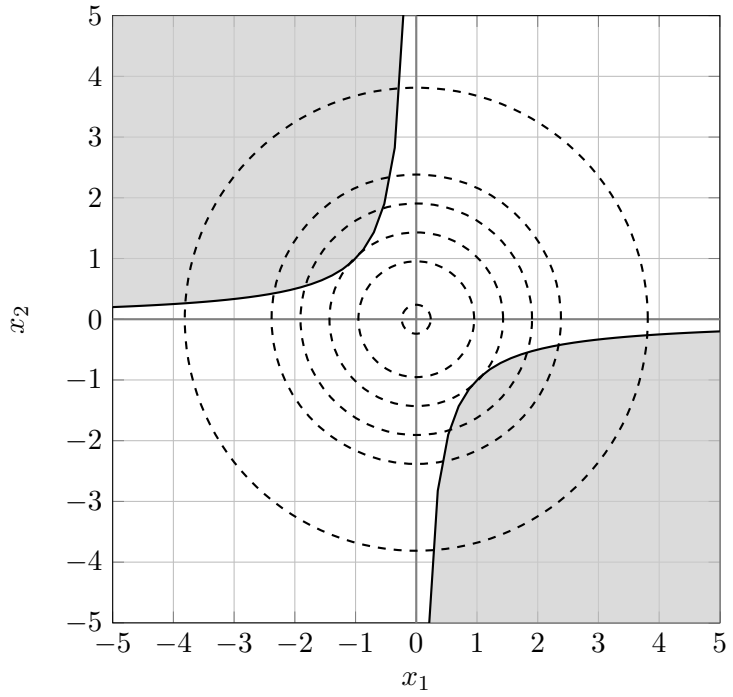

Figure 4.8: A plot showing the region of integration that evaluates the CDF $F_y(y)$ in Example Three, specifically for when $y = -1$.

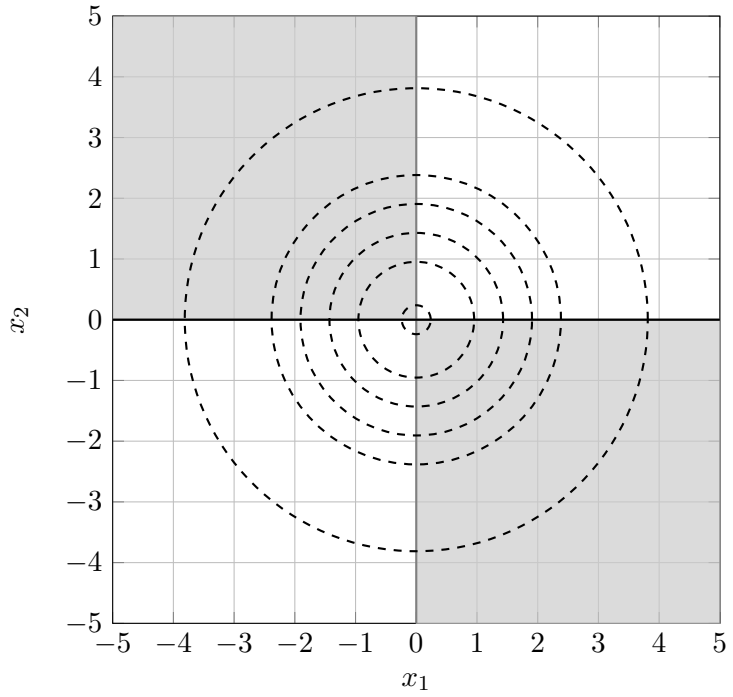

Figure 4.9: A plot showing the region of integration that evaluates the CDF $F_y(y)$ in Example Three, specifically for when $y = 0$.

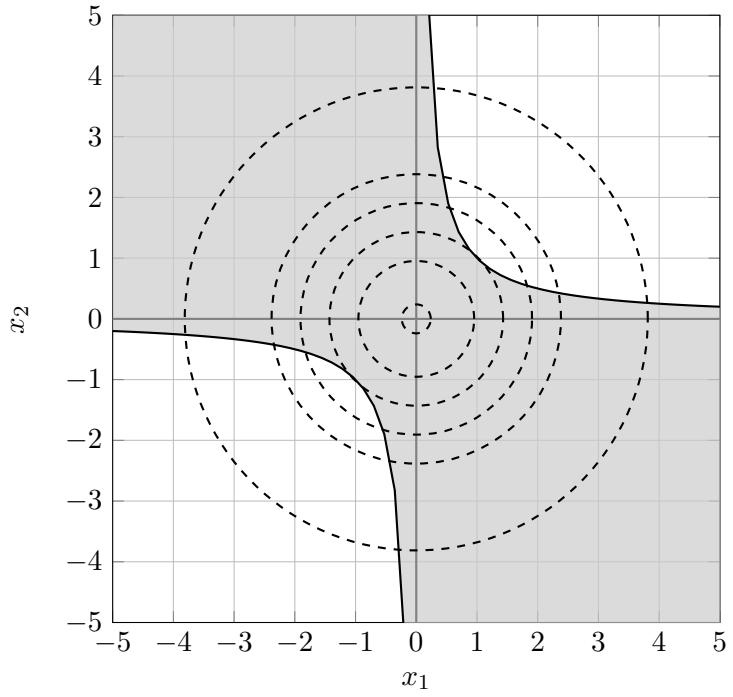

Figure 4.10: A plot showing the region of integration that evaluates the CDF $F_y(y)$ in Example Three, specifically for when $y = 1$.

That begin said, although the process is the same, the double integral in Equation 49 is far more complicated to evaluated analytically. Thus, to that end, we can use an already derived analytical expression, given by Weisstein (2003), for the PDF of the product of two normally

distributed variables that have zero means, which is

$$\begin{aligned} f_y(y) &= \int_{-\infty}^{\infty}\int_{\infty}^{\infty} \left(\frac{\exp\left(-\frac{x_1^2}{2\sigma_{x_1^2}}\right)}{\sqrt{2\pi}\sigma_{x_1}}\right)\left(\frac{\exp\left(-\frac{x_2^2}{2\sigma_{x_2^2}}\right)}{\sqrt{2\pi}\sigma_{x_2}}\right)\delta(x_1x_2-y)\,\mathrm{d}x_1\mathrm{d}x_2 \\ &= \frac{K_0\left(\frac{|y|}{\sigma_{x_1}\sigma_{x_2}}\right)}{\pi\sigma_{x_1}\sigma_{x_2}}. \end{aligned} \tag{50}$$

For the specific case where X_1 and X_2 are both distributed via the standard normal distribution ($\sigma_{x_1} = \sigma_{x_2} = 1$), we have

$$f_y(y) = \frac{K_0\left(|y|\right)}{\pi}. \tag{51}$$

Finally, we can plot $f_Y(y)$, as has been done in Figure 4.11.

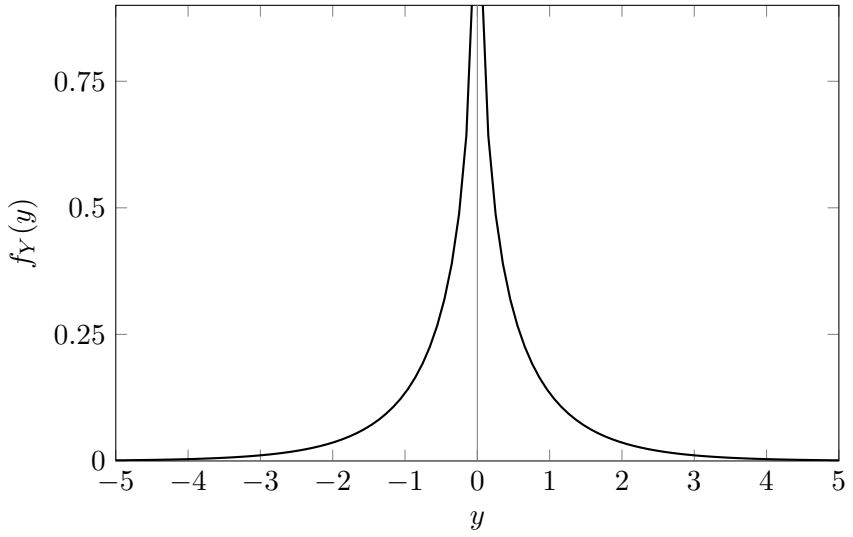

Figure 4.11: A plot showing the final PDF ,$f_Y(y)$, of the transformed random variable Y for Example Three.

4.3 Implementation in Haskell

4.3.1 Structure

Now that there is some familiarity with the mathematical procedure in Section 4.2, the implementation in Haskell can be made clear. It is worth noting that this implementation of the mathematical procedure in Haskell has been done in the nature of an editable script. Firstly, the overall structure and flow of information can be seen in Figure 4.12, where we know the transformation g, and the PDF of $\boldsymbol{X}$, $f_{\boldsymbol{X}}(\boldsymbol{x})$, (which may be joint, since $\boldsymbol{x}$ and $\boldsymbol{X}$ are vectors, hence these constructions may be multivariate).

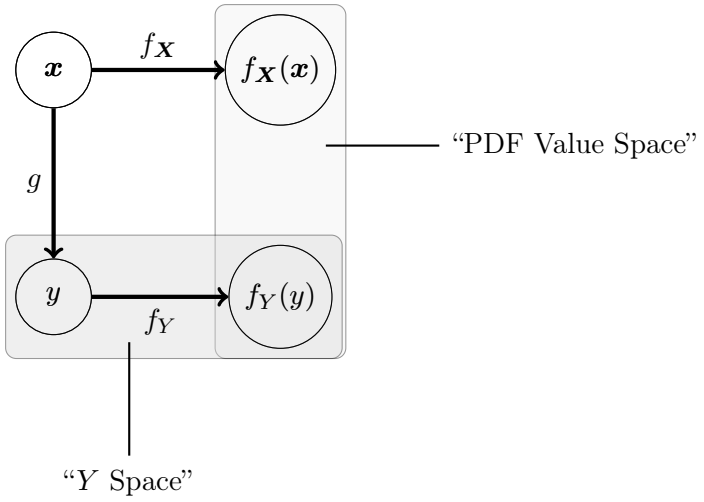

Figure 4.12: A figure showing a conceptual structure of the uncertainty propagation problem.

The important features of Figure 4.12 are the informal areas "Y space", and "PDF value space". These areas are ultimately areas where distinctions via context are important. For example, if we had values for $f_{\boldsymbol{X}}(\boldsymbol{x})$ and $f_Y(y)$, they would both be numbers in the set of positive real numbers. Hence, with those values alone, there is nothing to distinguish between them since $f_{\boldsymbol{X}}$ and f_Y may have the same rough codomain (set of outputs). Hence, context is needed to know that the value $f_Y(y)$ alone arises from the transformed random variable Y (within a kind of "Y space"). Furthermore, if we had some values of $\boldsymbol{x}$ and $f_{\boldsymbol{X}}(\boldsymbol{x})$ (where values of $\boldsymbol{x}$ represent the actual values of physical variables in the context of Buckingham Π terms), then both values may also be in the set of positive real numbers. Then again, more context is necessary to say that the values of $f_{\boldsymbol{X}}(\boldsymbol{x})$ alone arise from the PDF of $\boldsymbol{X}$ (within a kind of "PDF value space"). Likewise with values of y and $f_Y(y)$. Thus, there are these two important contexts for the values which we obtain, firstly of a Y space, and secondly of a PDF value space. And so, for a final example, a value of $f_Y(y)$ would first be embedded in the context of a PDF value space, but then also within the context of being within a Y space.

In order to represent these two contexts, the `Functor` type class was used, which was introduced in Section 2.6. Listing 14 shows the definitions of several basic custom algebraic data types. Lines 1, 11, and 21 are the core components of the definitions, each creating data types to represent the Y space (`YS a`), "CDF value space" (`CDF a`), and PDF value space (`PDF a`) respectively. Note that the CDF value space is new, but in exact like kind to the PDF value space. This is because in order to find $f_Y(y)$, we must first find $F_Y(y)$ (Equation 29).

```
1  data YS a = YS a
2    deriving (Show,Ord,Eq)
3
4  instance Functor YS where
5    fmap func (YS a) = YS (func a)
6
7  instance Applicative YS where
8    pure = YS
9    YS func <*> YS x = YS (func x)
10
11 data CDF a = CDF a
12   deriving (Show,Ord,Eq)
13
14 instance Functor CDF where
15   fmap func (CDF a) = CDF (func a)
16
17 instance Applicative CDF where
18   pure = CDF
19   CDF func <*> CDF x = CDF (func x)
20
21 data PDF a = PDF a
22   deriving (Show,Ord,Eq)
23
24 instance Functor PDF where
25   fmap func (PDF a) = PDF (func a)
26
27 instance Applicative PDF where
28   pure = PDF
29   PDF func <*> PDF x = PDF (func x)
30
31 fsCDFtoPDF (CDF a) = PDF a
32 fsPDFtoCDF (PDF a) = CDF a
33
34
35 type Transform = [Double] -> YS Double
36 type OriginalPDF = [Double] -> PDF Double
```

Listing 14: Custom algebraic data types for an uncertainty propagation implementation in Haskell.

Lines 35 and 36 of Listing 14 are type synonym definitions that make things a lot easier to read later on. They describe the type of a transformation, g, (`Transform`), and the type of the original known PDF function of $\boldsymbol{X}$, $f_{\boldsymbol{X}}$, (`OriginalPDF`). Each type represents a function that takes a list of `Double` values, and gives back a single `Double` value, embedded within the appropriate context (either in `YS`, or `PDF`). Lines 31 and 32 are simply utility functions that allow us to switch between the `PDF` and `CDF` contexts. All other lines that begin with the `instance` keyword are explicit definitions that allow `YS`, `CDF`, and `PDF` to become functors (and implement `fmap`), or in other words, to become instances of the `Functor` type class. Furthermore, `YS`, `CDF`, and `PDF` are also made to be instances of the `Applicative` type class. This type class is essentially a specific kind of functor, that allows functions also to be wrapped in a context, not just values (Bhargava, 2013). This becomes useful when we are sequencing many applications functions over a value within a context (within a functor). The `Applicative` type class also supplies a function called `pure` which is able to add a context (functor) onto a value or function.

Now the core data types and structures have been set up, the numerical algorithm can be begun. Listing 15 shows the setting up of an example with a two-dimensional domain to represent all

the values that are inputted into $f_{X_1,X_2}(x_1, x_2)$ with a transformation $g(x_1, x_2) = x_1 x_2$, where $X_1 \sim \mathcal{N}(10, 1)$ and $X_2 \sim \mathcal{U}(1, 3)$. In this example, `x1s` is a list of x_1 values from 0 to 20, spaced by 0.05, and `x2s` is a list of x_2 values from 0 to 5, spaced by 0.05 (thus both being of type `[Double]`). `dxs` and `xss` on line 3 and 7 simply package the spacing values, and the values of `x1s` and `x2s` themselves respectively.

```
1  dx1 = 0.05 :: Double
2  dx2 = 0.05 :: Double
3  dxs = [dx1,dx2]
4
5  x1s = [0,dx1..20]
6  x2s = [0,dx2..5]
7  xss = [x1s,x2s]
8
9  g :: Transform
10 g [x1,x2] = pure (x1 * x2)
11 g _ = pure 0
12
13 fPDF :: OriginalPDF
14 fPDF [x1,x2] = pure ((normpdf 10 1 x1) * (unifpdf 1 3 x2))
15 fPDF _ = pure 0
16
17 dy = YS 0.1
```

Listing 15: The editable portion of the Haskell script implementation of uncertainty propagation, which is the only section needed to be edited for a specific problem.

Lines 9 to 11 in Listing 15 define the transformation g, according to the `Transform` type in Listing 14, and lines 13 to 15 show the definition of the joint PDF $f_{X_1,X_2}(x_1, x_2)$, according to the `OriginalPDF` type in Listing 14. Finally, line 17 shows the definition of the numerical differential spacing used in order to calculate $f_Y(y)$ from $F_Y(y)$.

The final segment of this example program is the definition of the function `pushPDF` (Listing 16). This function performs numerical integration to find $F_Y(y)$ by evaluating (in pseudocode)

$$\texttt{pushCDF y} = \texttt{product [dxs]} \times \sum(\texttt{fPDF x1 x2}) \tag{52}$$

where `x1` and `x2` are all possible values within `x1s` and `x2s` respectively, that satisfy the condition `g [x1,x2] <= y`. Note that the types of the actual functions used to evaluate Equation 52 would mean that Equation 52 does not type match, hence why it must be a form of pseudocode. `product` in Equation 52 is a function that simply returns the product of each element in a list. The final step is the evaluation of the simple forward numerical differentiation (in pseudocode)

$$\texttt{pushPDF y} = \frac{(\texttt{pushCDF (y + dy)}) - (\texttt{pushCDF y})}{\texttt{dy}} \tag{53}$$

where `dy` is the predefined numerical differentiation spacing.

```
1 pushPDF :: OriginalPDF -> Transform -> YS Double -> YS (PDF Double)
2 pushPDF f g y = fmap fsCDFtoPDF $ (fmap . fmap) (/ ((\(YS u) -> u) dy)) dFy
3   where pushCDF yval = pure $ fsPDFtoCDF $ fmap (product dxs *) . fmap sum .
      sequenceA $ fCartProdFilt :: YS (CDF Double)
4           where fCartProdFilt = map f . filter (\xs -> g xs <= yval) $ sequence
      xss
5         dFy = (liftA2 . liftA2) (-) (pushCDF (liftA2 (+) y dy)) (pushCDF y) ::
    YS (CDF Double)
```

Listing 16: Definition of `pushPDF`.

Hence, we now have the full script to get a function that computes the value of $f_y(y)$, given a transformation g, and a known joint PDF $f_{\boldsymbol{X}}(\boldsymbol{x})$. The structure of the script is thus threefold by joining Listing 14, Listing 15, and Listing 16 together, where the only section that needs to be edited for any specific case is Listing 15.

What all of this added structure means in practical terms, is that when we use the `pushPDF` function, we must specify things in their correct context. We therefore have to write

```
GHCi> pushPDF fPDF g (YS 20)
YS (PDF 4.6669526438950504e-2)
```

where `fPDF` is the known PDF function, $f_{X_1,X_2}(x_1, x_2)$ (of type `OriginalPDF`), `g` is the transformation function, $g(x_1, x_2) = x_1 x_2$ (of type `Transform`), and `(YS 20)` is the y value (of type `YS Double`) where we wish to evaluate `pushPDF` at. These constrictions on the types provide a means to specify the context of every value, where the lack of any context (functor) represents a value of either x_1 or x_2. Hence, the result which is returned (`YS (PDF 4.6669526438950504e-2)`) is embedded in the nested contexts of being a PDF value, and being related to the random variable Y, and hence is a value of the PDF of Y.

4.3.2 Example One

In order to show further how the program works for individual cases, we can bring back the three examples from Section 4.2.

Example One was based around the formulation in Equation 30 which involved two known independent random variables with uniform distributions (each from 0 to 1). Hence, the domain of $f_{X_1,X_2}(x_1, x_2)$ needs only be evaluated for when both x_1 and x_2 are between 0 and 1, which is what lines 5 to 7 in Listing 17 define. The transformation g, which is involved is that of a product, which is what lines 9 to 11 define. Then, $f_{X_1,X_2}(x_1, x_2)$ is defined in lines 13 to 15, and finally `dy` is defined on line 17.

```
1  dx1 = 0.001 :: Double
2  dx2 = 0.001 :: Double
3  dxs = [dx1,dx2]
4
5  x1s = [0,dx1..1]
6  x2s = [0,dx2..1]
7  xss = [x1s,x2s]
8
9  g :: Transform
10 g [x1,x2] = pure (x2 * x1)
11 g _ = pure 0
12
13 fPDF :: OriginalPDF
14 fPDF [x1,x2] = pure ((unifpdf 0 1 x1) * (unifpdf 0 1 x2))
15 fPDF _ = pure 0
16
17 dy = YS 0.01
```

Listing 17: Implementation of Example One in Haskell (only the editable section of the script).

The result of this Haskell implementation for Example One can be found in Figure 4.13.

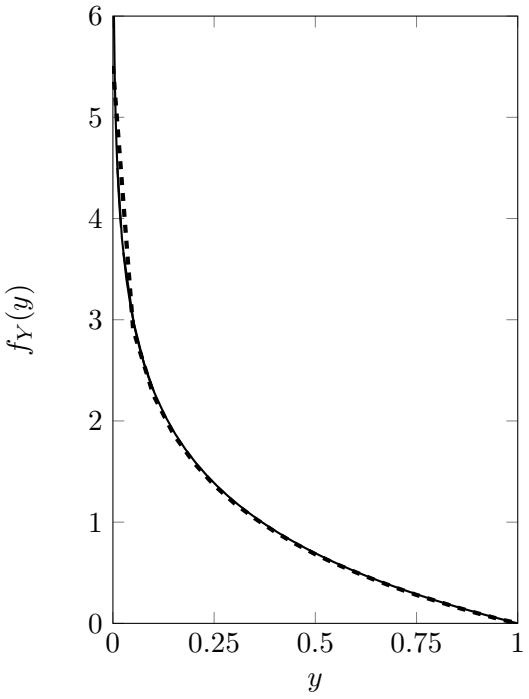

Figure 4.13: A plot comparing the final PDF ,$f_Y(y)$, of the transformed random variable Y attained by analytical means (solid black line) with the results of the Haskell Script implementation (dashed black line) for Example One.

Clear agreement with the analytical solution can be seen in Figure 4.13, which was defined in Equation 37. There is a slight difference as $y \to 0$ due to the numerical resolution, and also `pushPDF` gives a finite value for when $y = 0$, even though as $y \to 0$, $f_Y(y) \to \infty$. However, these things can be considered to be natural given the numerical nature of the integration and differentiation algorithms.

4.3.3 Example Two

Example Two was based around the formulation in Equation 38 and Equation 39 which involved two known independent random variables with uniform distributions (each from 0 to 1). Hence, this domain of $f_{X_1,X_2}(x_1, x_2)$ also needs only to be evaluated for when both x_1 and x_2 are between 0 and 1, which is what lines 5 to 7 in Listing 18 define. The transformation g is that of a quotient, which is what lines 9 to 11 define. Then, $f_{X_1,X_2}(x_1, x_2)$ is defined in lines 13 to 15, and finally `dy` is defined on line 17. Much of this is exactly the same as the implementation of Example One, where the only difference is the transformation g.

```
1  dx1 = 0.001 :: Double
2  dx2 = 0.001 :: Double
3  dxs = [dx1,dx2]
4
5  x1s = [0,dx1..1]
6  x2s = [0,dx2..1]
7  xss = [x1s,x2s]
8
9  g :: Transform
10 g [x1,x2] = pure (x2 / x1)
11 g _ = pure 0
12
13 fPDF :: OriginalPDF
14 fPDF [x1,x2] = pure ((unifpdf 0 1 x1) * (unifpdf 0 1 x2))
15 fPDF _ = pure 0
16
17 dy = YS 0.01
```

Listing 18: Implementation of Example Two in Haskell (only the editable section of the script).

The result of this Haskell implementation for Example Two can be found in Figure 4.14.

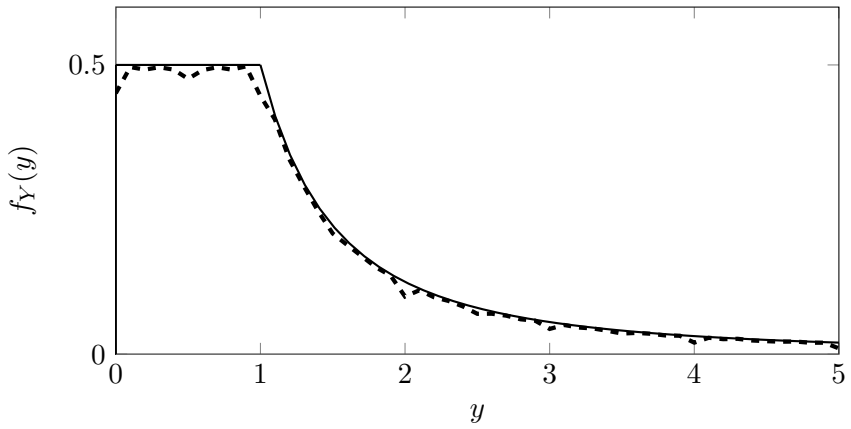

Figure 4.14: A plot comparing the final PDF ,$f_Y(y)$, of the transformed random variable Y attained by analytical means (solid black line) with the results of the Haskell Script implementation (dashed black line) for Example Two.

Again, clear agreement between with the analytical solution defined in Equation 46, can be found in Figure 4.14. There are however, signs of a lack of numerical resolution, in places, most likely due to the proximity of discontinuities at $y = 0$ and $y = 1$ in the analytical function defined in Equation 46.

4.3.4 Example Three

Example Three was based around the formulation in Equation 47 which involved two known independent random variables with standard normal distributions. Hence, the domain of $f_{X_1,X_2}(x_1, x_2)$ needs only be evaluated for when both x_1 and x_2 are between -3 and 3 (since outside of this range values are extremely close to 0), which is what lines 5 to 7 in Listing 17 define. The transformation g which is involved, is that of a product, which is what lines 9 to 11 define. Then, $f_{X_1,X_2}(x_1, x_2)$ is defined in lines 13 to 15, and finally `dy` is defined on line 17. Again,

much may seem unchanged, especially compared to Example One, but notable differences are the domain ranges (`xss`), and the known joint PDF function (`fPDF`).

```
dx1 = 0.01 :: Double
dx2 = 0.01 :: Double
dxs = [dx1,dx2]

x1s = [-3,(-3+dx1)..3]
x2s = [-3,(-3+dx2)..3]
xss = [x1s,x2s]

g :: Transform
g [x1,x2] = pure (x2 * x1)
g _ = pure 0

fPDF :: OriginalPDF
fPDF [x1,x2] = pure ((normstdpdf x1) * (normstdpdf x2))
fPDF _ = pure 0

dy = YS 0.01
```

Listing 19: Implementation of Example Three in Haskell (only the editable section of the script).

The result of this Haskell implementation for Example Three can be found in Figure 4.15.

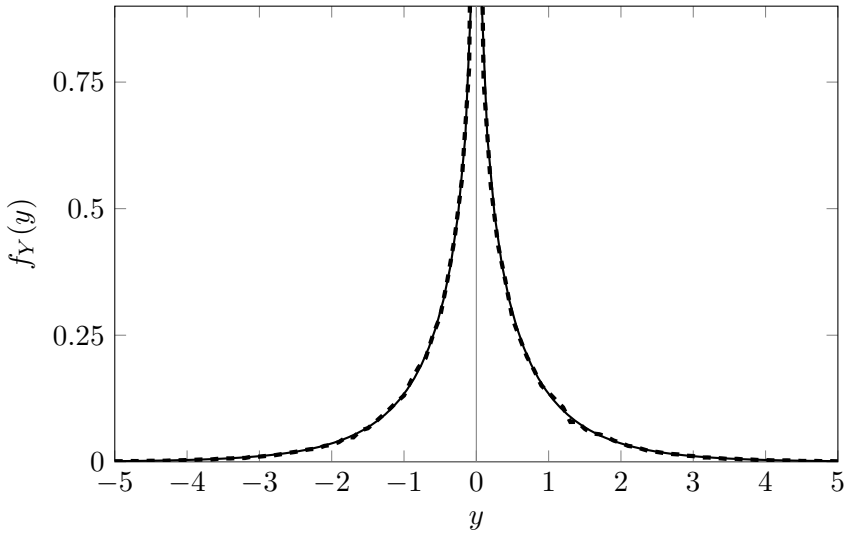

Figure 4.15: A plot comparing the final PDF ,$f_Y(y)$, of the transformed random variable Y attained by analytical means (solid black line) with the results of the Haskell Script implementation (dashed black line) for Example Three.

In this final example, again, very nice agreement can be found in Figure 4.15 with the analytical solution defined in Equation 51. However, the same problem (though it cannot be seen in Figure 4.15) arises as in Example One, where even though $f_Y(y) \to \infty$ as $y \to 0$ in the analytical formulation, `pushPDF` gives a finite value. But, in the same way, this can be considered natural given the numerical algorithms at hand.

4.3.5 An Example with a Buckingham Π Term

Let us return to Equation 25, which was a Π term given as an example right at the beginning of this section, based on the phenomenon of a swinging pendulum, which we should be familiar with by now. To be clear, the specific Π term in Equation 25 (on the third row) was

$$\Pi_2 = l^{-1/2} g^{1/2} \tau. \tag{54}$$

Now, let us say again that we realised that the three variables in question, l, g, and τ, were actually known to be independent random variables with known distributions. Then, we wanted to find the nature of the probability distribution associated with the Π term. We would begin the formulation of the problem by stating the known random variables, X_1, X_2, and X_3, along with the known transformation g, as in

$$\begin{aligned} X_1 &\sim \mathcal{N}(15, 1), \\ X_2 &\sim \mathcal{N}(9.8, 1), \\ X_3 &\sim \mathcal{N}(5.4, 1), \\ Y = g(X_1, X_2, X_3) &= X_1^{-1/2} X_2^{1/2} X_3, \\ f_{X_1,X_2,X_3}(x_1, x_2, x_3) &= f_{X_1}(x_1) f_{X_2}(x_2) f_{X_3}(x_3), \end{aligned} \tag{55}$$

where X_1, X_2, and X_3 are the random variables associated with the physical variables l, g, and τ respectively. We can then implement this in Haskell via editing the relevant portion of the script, such that we obtain Listing 20. Since the representation of the discretised domain (`xss`), the functions `g` and `fPDF` have been packaged with lists, the Haskell implementation can be extended easily, by just adding more elements to those lists. This helps us also because the types of the functions `g`, `fPDF`, and `pushPDF` do not need to be changed since a value of type `[Int]` has the same type whether that list of integers is 15 elements or 15,000 elements long.

```
1  dx1 = 0.1 :: Double -- l
2  dx2 = 0.1 :: Double -- g
3  dx3 = 0.1 :: Double -- tau
4  dxs = [dx1,dx2,dx3]
5
6  x1s = [12,(12+dx1)..18] -- l
7  x2s = [7,(7+dx2)..13] -- g
8  x3s = [2,(2+dx3)..8] -- tau
9  xss = [x1s,x2s,x3s]
10
11 g :: Transform
12 g [x1,x2,x3] = pure ((x1**(-0.5)) * (x2**0.5) * x3)
13 g _ = pure 0
14
15 fPDF :: OriginalPDF
16 fPDF [x1,x2,x3] = pure ((normpdf 15 1 x1) * (normpdf 9.8 1 x2) * (
       normpdf 5.4 1 x3))
17 fPDF _ = pure 0
18
19 dy = YS 0.1
```

Listing 20: Implementation of a three-dimensional example in Haskell (only the editable section of the script) with a Π term from the pendulum example given throughout this paper.

The only fundamental differences found in Listing 20 compared to other examples is that lines 3 and 8 are extra definitions for the third dimension of the domain, and that `g` and `fPDF` have

to accommodate an extra element in their input lists. Besides these things, everything else is essentially the same.

The results of implementing this example in the Haskell script can be seen in Figure 4.16.

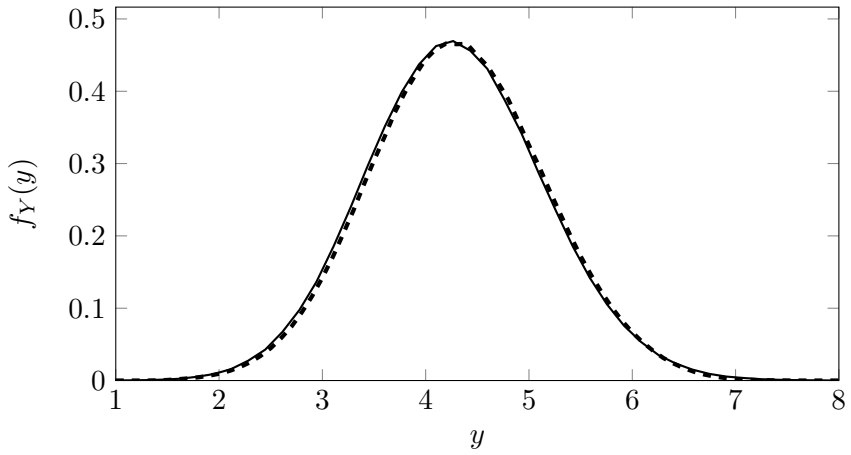

Figure 4.16: A plot comparing the final PDF ,$f_Y(y)$, of the transformed random variable Y attained by random sampling (solid black line) with the results of the Haskell Script implementation (dashed black line) for a Π term from the pendulum example.

Since no analytical solution to this problem could be done or found, the results of implementing this example in Haskell have been compared to histogram samples obtained from manipulating random samples of the distributions defined in Equation 55 using Python. Thus, we can say that the results from the Haskell script show clear agreement with this brute-force approach implemented in Python. As such, it is with a high degree of certainty that we can call this example's implementation in Haskell a success, in like manner to Example One, Two, and Three previously shown.

This example of the Π term taken from the scenario of the pendulum however is particularly apt, since it summarises the journey this paper sought to explore. $f_Y(y)$ shown in Figure 4.16 characterises the probability distribution of a Π term found using a certain Haskell implementation, where the Π term itself was also found using a Haskell implementation of the method of repeating variables. Thus, we have reached a (hopefully) satisfying end to this specific exploration.

5 Conclusion

5.1 Evaluation

The primary aim of this paper was to test if Haskell could produce structured programs in order to solve two prescribed problems within dimensional analysis. What have been built are two programmed implementations of solutions in Haskell, which are clearly structured, and solve the respective problems at hand.

Section 3 saw the creation of a program that could obtain Buckingham Π terms from a collection of physical variables relevant to a physical phenomenon. In Section 1.2, it was mentioned that a structured program is precisely one that is modular. Thus, the program seen in Section 3 is

primarily modular with respect to the composing of functions and high level modules. Section 4 saw the creation of a program that was modular in a slightly different way. The primary kind of modularity used was that afforded by Haskell's type system. The custom algebraic data types split up the domains of the problem in order to make it clear in what context different functions were operating.

In terms of the algorithms which were used, most if not all were rather straight forward, and so very little can be said of the efficiency of the programs that have been made. Nonetheless, particular concern must be made apparent, especially with regard to the Haskell implementation solving uncertainty propagation within Π terms (Section 4). The extremely basic numeric algorithms for integration and differentiation has meant that resolution of the results has been a potent issue. Furthermore, the time it takes to run, and the memory it uses are both considerable drawbacks. However, this paper did not seek to reconcile such concerns.

Thus, with all that this paper has shown, it is incredibly difficult to say that Haskell cannot offer unique tools for structuring clear and concise solutions, which are at best awkwardly expressed in other languages, at least to solve these prescribed problems in dimensional analysis.

5.2 Next Steps

There is no reason to suggest at all that the structures used within the programs that were made are the clearest, or most concise. Though such an aim in its extremity was not in the scope of this paper, it is a natural next step to seek out what sort of structures may be the most effective. This is most certainly the case for Section 4, where the formal representation of uncertainty propagation (a pushforward distribution) was unable to be mirrored accurately in the Haskell implementation. But by mirroring such representational structures, Haskell might be found to offer structures that are even more suited to these problems, as well as other similar problems.

Since the numerical algorithms for integration and differentiation used in Section 4 were so basic, it would also be a natural next step to improve such numerical algorithms to make them more efficient and less resource consuming.

5.3 Note on Source Code

As it has been mentioned already, all source code can be found on https://github.com (Symonds Patel, 2023). The Haskell implementation of the method of repeating variables was made in the structure of a library, along with a small basic application that can write out Π terms to a file, taking textual input also from a file. The Haskell implementation of uncertainty propagation was created in the manner of an interactive script, where it needs to be edited and used through GHCi (Haskell's interactive environment). Furthermore, all Haskell implementations in this paper used external libraries which can be seen in the source code, and were built using Cabal.

Acknowledgements

The author would like to give many thanks to Dr John Craske for being an exceptional supervisor, and for making this paper possible in the first place. The author would also like to thank Prof. Ricardo M. S. Rosa for providing explanations with regard to the mathematical procedures involved in the method of repeating variables.

References

Aaby, A. A. (1998). Functional programming. https://www.cs.jhu.edu/~jason/465/readings/lambdacalc.html [Accessed 12th June 2023]

Allen, C., Moronuki, J., & Syrek, S. (2016). *Haskell programming from first principles*. Lorepub LLC. https://books.google.co.uk/books?id=5FaXDAEACAAJ

Barenblatt, G. I. (1996). *Scaling, self-similarity, and intermediate asymptotics*. Cambridge University Press. https://doi.org/10.1017/CBO9781107050242

Bhargava, A. Y. (2013). Functors, applicatives, and monads in pictures. https://www.adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html [Accessed 12th June 2023]

Contreras, I., Dyachenko, S., & Muncaster, B. (2018). Applied linear algebra - chapter 2: Vector spaces, linear transformations, and their applications - section 4: The four fundamental subspaces. https://faculty.math.illinois.edu/~icontrer/Lecture10.pdf [Accessed 17th March 2023]

Gorelli, M. (2021). What's a pushforward distribution? https://towardsdatascience.com/whats-a-pushforward-distribution-17758c1fd542 [Accessed 14th June 2023]

Hughes, J. (1989). Why Functional Programming Matters. *The Computer Journal*, *32*(2), 98–107. https://doi.org/10.1093/comjnl/32.2.98

Jones, S. L. P., & Wadler, P. (1993). Imperative functional programming. *ACM-SIGACT Symposium on Principles of Programming Languages*,

Karam, M., & Saad, T. (2021). Buckinghampy: A python software for dimensional analysis. *SoftwareX*, *16*, 100851. https://doi.org/https://doi.org/10.1016/j.softx.2021.100851

Launchbury, J., & Jones, S. L. P. (1995). State in haskell. *LISP and Symbolic Computation*, *8*, 293–341.

Lu, Y. M. (2018). Topic 5: Functions of multivariate random variables. http://www.ece.tufts.edu/~maivu/ES150/5-mrv_func.pdf [Accessed 9th June 2023]

Rosa, R. M. S. (2021). Buckingham-pi theorem and the unitfulbuckinghampi.jl package. https://rmsrosa.github.io/blog/2021/05/unitfulbuckinghampi/ [Accessed 2nd March 2023]

Singh, I., Palakkandy, A., & Lima, F. (2018). Fourier analysis of nonlinear pendulum oscillations. *Revista Brasileira de Ensino de Física*, *40*. https://doi.org/10.1590/1806-9126-rbef-2017-0151

Sonin, A. A. (2001). The physical basis of dimensional analysis. http://web.mit.edu/2.25/www/pdf/DA_unified.pdf [Accessed 13th March 2023]

Strang, G. (2006). *Linear algebra and its applications* (Fourth edition, International Student Edition.). Thomson Brooks/Cole.

Symonds Patel, J. (2023). *Functional Dimensional Analysis for Engineers Using Haskell - Source Code* (Version 1.0.0). https://github.com/jlsymondspatel/Functional-Dimensional-Analysis-for-Engineers-Using-Haskell/

Weisstein, E. W. (2003). Normal product distribution from mathworld – a wolfram web resource. https://mathworld.wolfram.com/NormalProductDistribution.html [Accessed 11th June 2023]

Zohuri, B. (2015). *Dimensional analysis and self-similarity methods for engineers and scientists*. Springer Cham. https://doi.org/10.1007/978-3-319-13476-5