FORMALIZING DIMENSIONAL ANALYSIS USING THE LEAN THEOREM PROVER

A PREPRINT

Maxwell P. Bobbin[1], Colin Jones[1], John Velkey[1], and Tyler R. Josephson[1,2]

[1]Department of Chemical, Biochemical, and Environmental Engineering, University of Maryland Baltimore County,
1000 Hilltop Circle, Baltimore, MD 21250
[2]Department of Computer Science and Electrical Engineering, University of Maryland Baltimore County,
1000 Hilltop Circle, Baltimore, MD 21250

ABSTRACT

Dimensional analysis is fundamental to the formulation and validation of physical laws, ensuring that equations are dimensionally homogeneous and scientifically meaningful. While programming languages and computer algebra systems have long incorporated type systems or symbolic tools to track units, these implementations rarely provide formal guarantees about correctness. In this work, we use the Lean 4 theorem prover to formalize the mathematics of dimensional analysis. We define physical dimensions as mappings from base dimensions to exponents, prove that they form an Abelian group under multiplication, and implement derived dimensions and dimensional homogeneity theorems. Building on this foundation, we introduce a definition of physical variables that combines numeric values with dimensions, extend the framework to incorporate SI base units and fundamental constants, and implement the Buckingham Pi Theorem. Finally, we demonstrate the approach on an example: the Lennard-Jones potential, where our framework enforces dimensional consistency and enables formal proofs of physical properties such as zero-energy separation and the force law. This work establishes a reusable, formally verified framework for dimensional analysis in Lean, providing both a foundation for future libraries in formalized science and a pathway toward scientific computing environments with built-in guarantees of dimensional correctness.
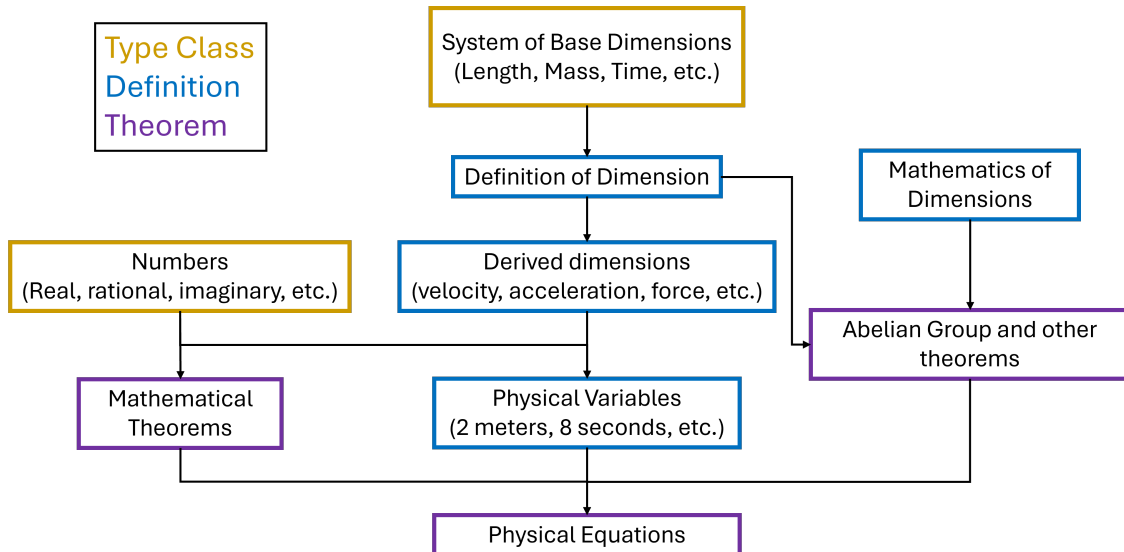
1 Introduction

Dimensions are fundamental to the way we observe, measure, and experiment with the world around us. Physical variables combine numbers and dimensions to describe the real world, and units provide the reference to communicate our observations. All of them are quintessential to engineering and scientific applications. Beyond giving us the ability to verbalize the universe, physical variables contain intrinsic properties that dictate how the formal science of mathematics can explain reality. Joseph Fourier first noted that physical variables can be grouped because they share the same dimension [1], meaning that they can be compared, and likely describe a common phenomenon. Mathematically speaking, physically valid relations (i.e. relations which describe the real world) can only be those whose dimensions on each side of the relationship are the same. This is the property of dimensional homogeneity and the foundation of scientific mathematics.

Dimensional analysis is the initial tool used to ensure the dimensional homogeneity of formulae [2], and has been studied in programming languages using type theory since the 1970s [3, 4]. Dimensions and units have been coded in common programming languages with type checking, like Fortran[5], Ada[6], C++[7], Standard ML[8], and Haskell [9]. Symbolic programming languages and computer algebra systems (CAS), like SymPy [10], F#[11, 12], and more [13, 14] have also been used to create programs to alleviate the process, with a focus on the Buckingham Pi theorem. In parallel, the mathematical properties of dimensions and physical variables have been studied [15–18], determining, for instance, that dimensions form an Abelian group for multiplication [1, 17]. However, the code created to implement physical variables and tools, like the Buckingham Pi theorem, has yet to be implemented in a way that formally encompasses the properties of dimensional analysis and the fact that it forms an Abelian group.

Proof assistants, also known as interactive theorem provers [19–21], are a type of programming language that would allow the possibility of defining dimensional analysis and verifying the correctness of the definition through formal proofs. This would result in the same usability as these other programs, but with the added reliability that comes from a formal environment. In this paper, we present a definition of physical dimensions using the Lean 4 theorem prover and a derivation of the properties of dimensions, such as the formation of an Abelian group under multiplication. Unlike unit systems in other programs, this implementation is built upon the Lean 4 kernel, ensuring that any theorem written is logically correct, so long as it can be parsed by Lean 4.

Our previous paper has shown the basics of using Lean for scientific applications, where fields such as thermodynamics and kinematics can be formally defined and verified [22]. That work introduces formal mathematics and Lean for scientific applications. Tooby-Smith has shown applications of Lean for high energy physics (HEPLean) [23] and digitalizing results of physics (PhysLean) [24]. Ugwuanyi et. al. have shown how Lean can be used for scientific computation with executable Lean code to calculate the potential energy between molecules with the Lennard-Jones potential [25]. We continue that work here by formalizing the mathematics of dimensional analysis in science and engineering, which should find applications both in the development of libraries for formalized science, as well as in the development of scientific computing software with formal guarantees of correctness.

This paper is structured as follows: in Section 3, we provide an overview of the nature and mathematics of physical variables and dimensions. Then, in Section 4, we show the implementation of dimensions (4.1), the mathematical properties of dimensions (4.2), that dimensions form an Abelian group (4.3), derived dimensions and dimensional homogeneity (4.4), and physical variables and units (4.5) in Lean.

2 Methods

We used Lean 4, an interactive theorem prover using its mathematical library, Mathlib, version 4.15.0-rc1. Our proofs are hosted on GitHub. Each code block has a *source* button that links to the code line in the GitHub repository.

3 Overview of Physical Variables and Dimensions

A physical quantity is a measure of a system containing a numeric, pure value and a reference unit. A physical variable is a representation of a physical quantity. For example, in $v = 10$ m/s, the velocity, v is the physical variable that represents the idea of an object moving, "10 m/s" is the physical quantity, 10 is the value, and m/s is the reference unit. While the physical quantity "10 m/s" is tied to its specified units, the physical variable v is more abstract, and has the added bonus of being invariant to the choice of units [26]. The dimensions of v are length/time, these are also invariant to the choice of units.

2

3.1 Physical Variables and Their Mathematical Properties

Formally, we can think of a physical variable, $\mathcal{P}$, as a type containing both a value, $\mathcal{V}$, and a dimension, $\mathcal{D}$, i.e. equation 1.

$$\mathcal{P} = \langle \mathcal{V}, \mathcal{D} \rangle \tag{1}$$

When an operator, $\star$, acts on physical variables, it interacts with the value and the dimension of the variable separately, as shown in Eq. 2.

$$\mathcal{P}_i \star \mathcal{P}_j = \langle \mathcal{V}_i \star \mathcal{V}_j, \mathcal{D}_i \star \mathcal{D}_j \rangle \tag{2}$$

From this, we can understand the mathematical analysis of physical equations to be two-part: an analysis of the values and an analysis of the dimensions. Since Mathlib has developed almost all of the mathematical analysis needed for the scientific analysis of the values, we shall focus on dimensional analysis to define physical variables.

3.2 Dimensions and their Mathematical Properties

The International System of Units, SI units, defines a dimension as a product of International System of Quantities (ISQ) base dimensions raised to a rational[1] power [27]. There are seven ISQ base dimensions: *Length (L), Time (T), Mass (M), Electric current (I), Temperature (θ), Amount of Substance (N), and Luminous Intensity (J)* [28]. Any dimension represented by the ISQ base dimensions can be constructed using Eq. 3.

$$\mathcal{D} = L^a T^b M^c I^d \theta^e N^f J^g \quad a, b, c, d, e, f, g \in \mathbb{Q} \tag{3}$$

Thus, length can be defined by setting $a = 1, b = 0, c = 0, ...$ and velocity can be defined as $a = 1, b = -1, c = 0, d = 0, ...$, and so on for any other dimension. Note that we have just defined two different versions of "length". The first was the base dimension *Length*, and the second was the actual dimension length. A base dimension is used to define a system and construct other dimensions. It will be denoted by capitalizing the first letter and italicizing. Dimensions will be denoted in lower and normal case[2].

$$velocity(L) = 1 \quad velocity(T) = -1 \quad velocity(M) = 0 \tag{4}$$

Eq. 3 has flaws as a general definition for a dimension. First, it's incomplete because it doesn't account for other base dimensions, like *Currency*, *Number of People*, etc. Second, it is cumbersome and inflexible because it requires us to account for every base dimension, even if our system doesn't use it. Finally, it limits us to rational numbers for exponents, which limits us to rational numbers for powers of physical variables. Therefore, a better definition needs to be able to include new base dimensions easily, allow the user to specify which base dimensions they want to consider, and allow flexibility in the exponent type.

Operation	Exponent Manipulation/Result
$a * b$	Exponents sum.
a/b	Exponents subtract.
$a + a$	Exponents stay the same
$a - a$	
$Log(a/a)$	All exponents zero (dimensionless number).

Table 1: Effects of Operations on Dimension Exponents

Therefore, we will define a dimension as a mapping of a base dimension, $\mathcal{B}$, to an exponent, $\mathcal{E}$, Eq. 5. We also define $\mathcal{E}$ as a type that forms a commutative ring. Since $\mathcal{E}$ forms a commutative ring, the simplest numerical type $\mathcal{E}$ can represent is the integers.

$$\mathcal{D} = \mathcal{B} \to \mathcal{E} \tag{5}$$

As an example, consider a system with three relevant base dimensions: *Length*, *Time*, and *Mass* (a fundamental system in kinematics). $\mathcal{B}$ has three elements L, T, and M. We can define length as the function that returns 1 if the base dimension is *Length* and 0 everywhere else (Eq. 6). Velocity is defined in a similar way (Eq. 7).

$$length(\mathcal{X}) := \begin{cases} 1 & \mathcal{X} = L \\ 0 & \mathcal{X} \neq L \end{cases} \tag{6}$$

$$velocity(\mathcal{X}) := \begin{cases} 1 & \mathcal{X} = L \\ -1 & \mathcal{X} = T \\ 0 & \mathcal{X} \neq L \neq T \end{cases} \tag{7}$$

---

[1]Rational numbers are a set of numbers that can be represented as the ratio of a natural number and an integer. Examples of rational numbers are $\frac{1}{2}, -2, -\frac{3}{4}$. Real numbers like $\sqrt{2}$ and π are excluded as powers in dimensional analysis.

[2]While we will try to avoid it, if a dimension starts a sentence, it will be uppercase, which is why we also italicize base dimensions to avoid confusion.

Base dimensions can then be used as inputs into a dimension to determine the respective exponent. Eq. 4 shows an example of indexing the dimension velocity with our system of three base dimensions.

Another, and much more convenient, way to define the velocity dimension would be as the quotient of length and time. Once the math of arithmetic operators is defined in Lean, all derived dimensions will be defined using these operators rather than as step-wise functions. All of these operators involve manipulating the exponent [29] that is returned when indexed by a base dimension, just like the manipulation commonly shown in Eq. 3. Table 1 gives an overview of the exponent manipulation for common operators.

As stated above, dimensions form an Abelian group, also known as a commutative group. The properties of an Abelian group are presented in Table 2.

Associativity	$\forall\, a\, b\, c, (a * b) * c = a * (b * c)$
Identity element	$\forall\, a, \exists\, e, a * e = e * a = a$
Inverse element	$\forall\, a, \exists\, b, a * b = b * a = e$
Commutativity	$\forall\, a\, b, a * b = b * a$

Table 2: The mathematical properties of an Abelian group for multiplication. The variables *a* and *b* are any variable from the set for which the Abelian group applies too. The variable *e* is a specific variable called the identity element. The symbol $\forall a$ reads "for all a" and refers to all elements of the set. The symbol $\exists a$ means "there exists an a" and refers to at least one element of the set.

4   Implementation in Lean

Now we shall turn our attention to defining physical variables and dimensions in Lean. We start with dimensions, which involves defining: how operators act on dimensions, integrating them with Lean's type classes, proving dimensions form an abelian group, creating derived dimensions and dimensional homogeneity theorems, and defining the Buckingham Pi Theorem. Then, we use our definition of dimensions to create physical variables and units, which uses the math defined for dimensions.

4.1   Definition of Dimensions and Base Dimensions

The dimension type is defined in Lean based on Equation 5. It takes two parameters: B, representing the system of base dimensions, and E, the type used for exponents. The type dimension is then a mapping from base dimensions to exponents:

```
/-- A dimension is a mapping from each base dimension to the exponent. -/            (source)
def dimension (B : Type u) (E : Type v) [CommRing E] := B → E
```

By leaving B abstract, we allow the same formalism to apply to various systems of base dimensions (e.g., ISQ, mechanical, etc.). For example, consider the kinematic system mentioned previously. This can be defined in Lean as:

```
inductive KinematicSystem
| Length | Time | Mass |
```

Here, inductive is Lean's keyword for defining inductive data types. *KinematicSystem* introduces three base dimensions: *Length*, *Time*, and *Mass*. We could also define a second base dimension system to just consider space and time:

```
inductive SpatialTemporalSystem
| Length2 | Time2 |
```

This system includes elements Length2 and Time2, which correspond to Length and Time in the kinematic system. We intentionally name them differently to emphasize that Lean treats these as distinct types, even though they conceptually represent the same base dimensions. As a result, Lean does not assume any connection between Length and Length2. To reconcile different systems referring to the same physical base dimension, we introduce type classes. To represent the existence of a base dimension *Length*, we define a class HasBaseLength as follows:

```
/-- Type class for base Length-/                                              (source)
class HasBaseLength (B : Type u) where
[dec : DecidableEq B]
Length : B
```

This class asserts two things: that equality between elements of the base dimension type B is decidable, and that B includes an element *Length*. By defining instances of this class, we can relate different systems to the same conceptual base dimension:

```
/-- Base Length instance for Kinematic System -/
instance : HasBaseLength KinematicSystem :=
{ dec := KinematicSystem.DecidableEq, Length := KinematicSystem.Length }

/-- Base Length instance for Spatial-Temporal System -/
instance : HasBaseLength SpatialTemporalSystem :=
{ dec := SpatialTemporalSystem.DecidableEq, Length := SpatialTemporalSystem.Length2 }
```

Now, both systems are unified under the shared concept of a *Length* base dimension. This design allows us to write generic code and theorems over arbitrary systems, as long as they satisfy the relevant type class constraints. Our current implementation defines a class for all seven ISQ base dimensions, and a currency base dimension to illustrate other potential base dimensions.

```
/-- Seven base dimensions from ISQ -/                                         (source)
-- Length defined above
class HasBaseTime (E : Type u) where
  [dec : DecidableEq B]
  Time : E

class HasBaseMass (B : Type u) where
  [dec : DecidableEq B]
  Mass : B

class HasBaseAmount (B : Type u) where
  [dec : DecidableEq B]
  Amount : B

class HasBaseCurrent (B : Type u) where
  [dec : DecidableEq B]
  Current : B

class HasBaseTemperature (B : Type u) where
  [dec : DecidableEq B]
  Temperature : B

class HasBaseLuminosity (B : Type u) where
  [dec : DecidableEq B]
  Luminosity : B

/-- Base dimension for Currency -/
class HasBaseCurrency (B : Type u) where
  [dec : DecidableEq B]
  Currency : B
```

4.2   Defining the Mathematics of Dimensions

Multiplication and division are defined as a function that takes in two elements and outputs an element of the same type. Then, the *instance* command is used to globally unify the definition with the respective class. This makes sure that, across Lean, all theorems are talking about the same addition, same multiplication, etc. It also allows us to access the

symbols used for these operations ($+, -, *, /, etc$). Multiplication and division for dimensions are defined in Lean as:

```
/-- Definition of multiplication for dimensions -/                          (source)
protected def mul {B : Type u} {E : Type v} [CommRing E] : dimension B E → dimension B E →
    dimension B E
| a, b => fun i => a i + b i
```

```
/-- Definition of division for dimensions -/                                (source)
protected def div {B : Type u} {E : Type v} [CommRing E] : dimension B E → dimension B E →
    dimension B E
| a, b => fun i => a i - b i
```

The definition uses *fun* to construct a new dimension by indexing through each base dimension of the input dimensions and adding or subtracting the exponent value for each base dimension. Raising a dimension to a power is defined as a function that takes in a dimension and a value (the value of the power) and outputs a dimension.

```
/-- Definition of powers for dimensions -/                                  (source)
protected def pow {B : Type u} {E : Type v} [CommRing E]: dimension B E → E → dimension B E
| a, n => fun i => n * (a i)
```

Even though the math has been defined, writing theorems would be cumbersome because we don't have access to the mathematical operators (which will make it easier to read our code and also allow the tactics in Lean to use the definitions). To access the operator symbols, the math that was defined needs to be globally harmonized with its respective classes. This is the same idea used for the base dimension classes. This ensures that, across Lean, all theorems are talking about the same operators. Thus, general theorems about operators can be written and applied to specific cases, like real numbers. For multiplication and division, this looks like:

```
/-- Unifying multiplication and division definitions with respective type class -/  (source)
instance {B : Type u} {E : Type v} [CommRing E] : Mul (dimension B E) := ⟨dimension.mul⟩
instance {B : Type u} {E : Type v} [CommRing E] : Div (dimension B E) := ⟨dimension.div⟩
```

The rest of the operators are instantiated in the same way.

Defining addition and subtraction takes special care when it comes to dimensional analysis. Two dimensions can be added (or subtracted) only if they are the same dimension. The result is the same dimension. When two scalars that are the same are added together, the result is twice the original number, i.e. $a + a = 2a$. However, for dimensional analysis, the result is just the original dimension, i.e. $(a : dimension) + a = a$. In the same way, the subtraction of two dimensions should yield the same dimension, $(a : dimension) - a = a$, instead of zero. Addition, like multiplication, is defined as a function that takes in two dimensions and outputs a dimension. However, there is no manipulation of exponents. The definition of addition in Lean is achieved using *Classical.epsilon*, which is the Hilbert epsilon function. Bell [30] and Wirth [31] both give detailed accounts of the epsilon function, its relation to the axiom of choice, and formal proofs. This gives a formal way of saying *if* $a = b$, $a + b = a$. In Lean, the definition of addition looks like:

```
/-- Definition of addition for dimensions-/                                 (source)
protected noncomputable def add {B : Type u} {E : Type v} [CommRing E] : dimension B E →
    dimension B E → dimension B E := Classical.epsilon $ fun f => ∀ a b, a = b → f a b = a
```

The *noncomputable* tag is used to signify that the definition cannot be compiled by Lean for the use of the *#eval* command, which is a command to evaluate objects. For instance, *#eval 2 + 2* would return *4* in the *Lean infoviewer*. This definition is then unified with the addition class so the + operator can be used. Substitution is defined in the exact same way and unified with the substitution class.

4.3 Proving Dimensions Form an Abelian Group

Now, with the math defined for dimensions, we can prove that the Abelian group properties hold. In Lean, the Abelian group, called *CommGroup*, is defined below (note that the numbers are used for the caption to explain each line, but do

not actually appear in the Lean code):

```
/-- Type class for commutative groups in Mathlib -/
class CommGroup (G : Type u) : Type u
(1)   mul : G → G → G
(2)   mul_assoc (a b c : G) : a * b * c = a * (b * c)
(3)   one : G
(4)   one_mul (a : G) : 1 * a = a
(5)   mul_one (a : G) : a * 1 = a
(6)   npow : ℕ → G → G
(7)   npow_zero (x : G) : Monoid.npow 0 x = 1
(8)   npow_succ (n : ℕ) (x : G) : Monoid.npow (n + 1) x = Monoid.npow n x * x
(9)   inv : G → G
(10) div : G → G → G
(11) div_eq_mul_inv (a b : G) : a / b = a * b⁻¹
(12) zpow : ℤ → G → G
(13) zpow_zero' (a : G) : DivInvMonoid.zpow 0 a = 1
(14) zpow_succ' (n : ℕ) (a : G) : DivInvMonoid.zpow (↑n.succ) a = DivInvMonoid.zpow (↑n) a * a
(15) zpow_neg' (n : ℕ) (a : G) : DivInvMonoid.zpow (Int.negSucc n) a = (DivInvMonoid.zpow
       (↑n.succ) a)⁻¹
(16) inv_mul_cancel (a : G) : a⁻¹ * a = 1
(17) mul_comm (a b : G) : a * b = b * a
```

(1) The multiplication operator (defined as a function that takes in two elements and outputs an element). (2) The fact that multiplication is associative (a*b)*c=a*(b*c). (3) The identity element (it is called one, because one is most commonly the identity element for practical representations of numbers). (4) & (5) the identity element section from the Abelian group definition (Table 2). (6) The natural power operator. (7) The fact that $x^0 = 1$. (8) The fact that $x^{n+1} = x * x^n$. (9) The inverse operator. (10) The division operator. (11) The fact that $a/b = a * b^{-1}$. (12) The Integer power operator. (13) The same as (7), but for an integer. (14) The same as (8) but for an integer. (15) The fact that $a^{-n} = (a^n)^{-1}$. (16) The inverse element from the Abelian group definition (Table 2). (17) The commutativity property of the Abelian group definition (Table 2).

At first glance, *CommGroup* appears to be more in-depth than the Abelian group. However, it doesn't define anything outside of the Abelian group. Instead, it has to talk about each case (the four parts to the table, plus division to make programming easier [3] and the operators needed (multiplication, division, inverse, npow, and zpow, along with the identity element). Next, we will walk through the proofs done in Lean to show each of these properties. Finally, we show the instance command that proves to Lean that *dimension* forms an Abelian group.

The first property we show is that multiplication is commutative (17), $(a * b = b * a)$. In Lean, this theorem looks like:

```
/-- Theorem proving that multiplication of dimensions is commutative -/          (source)
protected theorem mul_comm {B : Type u} {E : Type v} [CommRing E] (a b : dimension B E) : a * b =
    b * a := by
  simp only [mul_def']
  funext
  rw [add_comm]
```

Next, multiplication by the identity element (4) is shown. The other version (5) can be shown by using the *mul_comm* theorem that was just proven. This theorem answers the question of what the identity element for dimensions is. Since multiplication involves adding the values of the exponents of the two dimensions being multiplied, multiplying by a dimensionless dimension results in adding zero to all the values, which preserves the original dimension. In Lean, this looks like this:

---

[3]The division and the inverse operator go hand in hand, so it is natural to bring in division when the inverse operation is talked about. That is why, even though Table 2 doesn't explicitly mention the division operator, *CommGroup* still talks about it.

```
/-- Theorem proving that one (dimensionless) multiplied by a dimension equals          (source)
the original dimension -/
protected theorem one_mul {B : Type u} {E : Type v} [CommRing E] (a : dimension B E) : 1 * a = a
    := by simp only [one_eq_dimensionless,
  dimensionless_def', Function.const_zero, mul_def', Pi.zero_apply, zero_add]
```

Next, the associativity of multiplication (2) is shown:

```
/-- Theorem proving that multiplication of dimensions is associative -/                (source)
protected theorem mul_assoc {B : Type u} {E : Type v} [CommRing E] (a b c : dimension B E) : a *
    b * c = a * (b * c) := by
  simp only [mul_def']
  funext
  rw [add_assoc]
```

The final property we show is the inverse element (16) for dimensions. Another part that is added in the *CommGroup* definition is the relationship between division and multiplying by the inverse ($a/b = a * b^{-1}$). Both of those are shown below. Note that in the *mul_left_inv* proof, the number 1 is used in place of *dimensionless*. The number 1 is the identity element operator just like * is the multiplication operator, and comes from unifying *dimensionless* with *one*.

```
/-- Theorem proving that the inverse of a dimension multiplied by the                  (source)
same dimension yields dimensionless (one) -/
protected theorem mul_left_inv {B : Type u} {E : Type v} [CommRing E] (a : dimension B E) : a⁻¹ *
    a = 1 := by
  simp
  funext
  simp
```

```
/-- Theorem proving the relation between division and multiplication by an inverse -/  (source)
protected theorem div_eq_mul_inv {B : Type u} {E : Type v} [CommRing E] (a b : dimension B E) :
    a / b = a * b⁻¹ := by
  simp
  funext
  rw [sub_eq_add_neg]
```

In all of these proofs, tactics like *simp* and *funext* were used extensively along with a couple of other lemmas, whose proofs were not shown, but available on GitHub. The ability to use tactics to simplify the proof process is a result of deriving a large set of helper lemmas attached to the *simp* tactic that automate the tedious process of reverting back to the base definition of dimension and applying proofs to function mappings.

The final step is to use the *instance* command to prove to Lean that dimensions form an Abelian group. To do this, we must give the proof for each part of the Abelian group. Since we have already proven the individual theorems, we just have to reference the theorem. In Lean:

```
/-- Instance proving that dimensions form a commutative (abelian) group -/        (source)
instance {B : Type u} {E : Type v} [CommRing E] : CommGroup (dimension B E) where
  mul := dimension.mul
  div := dimension.div
  inv a := dimension.pow a (-1)
  mul_assoc := dimension.mul_assoc
  one := dimensionless B E
  npow n a := dimension.pow a ↑n
  zpow z a:= dimension.pow a ↑z
  one_mul := dimension.one_mul
  mul_one := dimension.mul_one
  mul_comm := dimension.mul_comm
  div_eq_mul_inv a := dimension.div_eq_mul_inv a
  inv_mul_cancel a := dimension.mul_left_inv a
  npow_zero := by intro x; funext x; simp
  npow_succ n a := by simp; funext x; rw [add_one_mul]
  zpow_neg' _ _ := by simp; rename_i x1 x2; funext x3; rw [← neg_add,neg_mul,add_comm]
  zpow_succ' _ _ := by simp; rename_i x1 x2; funext; rw [add_one_mul]
  zpow_zero' := by intro x; funext x; simp
```

4.4   Derived Dimensions and Dimensional Homogeneity Theorems

Now that all the math has been defined and unified with Lean's class system, regular dimensions can be defined, and theorems about the dimensional homogeneity of equations can be written and easily proved. As was mentioned above, the dimension length, Eq. 6, is defined as a function that evaluates to 1 at *Length* and 0 everywhere else. In Lean, this looks like,

```
/-- The dimension length -/                                                        (source)
def length (B : Type u) (E : Type v) [CommRing E] [HasBaseLength B] : dimension B E := Pi.single
    HasBaseLength.Length 1
```

The *[HasBaseLength B]* part requires that the system "has *Length*" as one of its base dimensions. The system could have other base dimensions, but the only important one for this definition is *Length*. The command *Pi.single HasBaseLength.Length 1* creates a function that is 1 at *HasBaseLength.Length* (the base dimension *Length*) and zero everywhere else. This approach is both flexible and expandable, allowing for new systems or base dimensions to be added in a modular fashion without modifying existing logic or definitions.

The dimension time can be defined in the same way, except it requires that B "has *Time*" instead of *Length*.

```
/-- The dimension time -/                                                          (source)
def time (B : Type u) (E : Type v) [CommRing E] [HasBaseTime B] : dimension B E := Pi.single
    HasBaseTime.Time 1
```

More advanced dimensions can be defined in a much more familiar way. While we could use the *Pi* function to continue defining dimensions, we can instead use the primary dimensions just defined and the math defined in the previous section. The dimensions velocity and acceleration can be defined as:

```
/-- The dimension velocity -/                                                      (source)
abbrev velocity (B : Type u) (E : Type v) [CommRing E] [HasBaseLength B] [HasBaseTime B] :=
    length B E / time B E
/-- The dimension acceleration -/
abbrev acceleration (B : Type u) (E : Type v) [CommRing E] [HasBaseLength B] [HasBaseTime B] :=
    length B E / ((time B E) ^ 2)
```

In this example, B needs to have both *Time* and *Length* since it references the length and time definition. Since the division of two dimensions is defined as subtracting their exponents, this creates a function which is 1 at *Length*, -1 at

Time, and zero everywhere else. Another example is the Reynolds number:

```
/-- Dimension of the Reynolds number -/                                    (source)
abbrev reynolds_number (B : Type u) (E : Type v) [CommRing E] [HasBaseLength B] [HasBaseTime B]
    [HasBaseMass B] := mass_density B E * velocity B E * length B E / dynamic_viscosity B E
```

Finally, theorems about the dimensional homogeneity of equations can be written. For instance, this is what a theorem showing that acceleration is dimensionally homogeneous to velocity divided by time looks like in Lean.

```
/-- Theorem proving the relation between the dimension of acceleration, velocity, and time (source)
theorem accel_eq_vel_div_time {B E} [CommRing E][HasBaseLength B] [HasBaseTime B] : acceleration
    B E = velocity B E / time B E := by rw[acceleration,velocity,pow_two,div_div]
```

Another example is shown below, showing that the Reynolds number is dimensionless:

```
/-- Theorem proving that the Reynolds number is dimensionless -/           (source)
theorem reynolds_eq_dimless (B : Type u) (E : Type v) [CommRing E] [HasBaseLength B]
    [HasBaseTime B] [HasBaseMass B] : reynolds_number B E = dimensionless B E := by
  rw [reynolds_number,mass_density,volume,velocity,dynamic_viscosity, ← one_eq_dimensionless,
    div_eq_one]
  rw [mul_assoc,mul_comm (length B/time B),mul_div,pow_three,
  ← mul_div_assoc,mul_comm,← mul_div_assoc,mul_comm _ (length B E * length B E),
    mul_div_mul_comm,
  ← div_one (length B E * length B E),div_div_div_cancel_left,div_one,one_mul,div_div]
```

4.5 Physical Variables and Units

With dimensions fully defined, we can build on this to define physical variables. This is done as a structure with two parts: the value of the measurement and the dimension describing the kind of measurement.

```
/-- Definition of a physical variable -/                                   (source)
structure PhysicalVariable (B V) [Field V]   where
(value : V)
(dim : dimension B V)
```

Here, B is the type representing the system of base dimensions, and V is the type of the measured value. We also require the value type and exponent type to be the same, as it is easier to read and doesn't reduce the applicability. As shown in Equation 2, operators act on the value and dimension separately. The definition of multiplication is shown below:

```
/-- Definition of multiplication for physical variables -/                 (source)
def PhysicalVariable.Mul : B E →   def PhysicalVariable.Mul : B E → def PhysicalVariable.Mul :
    B E
| a,b => def PhysicalVariable.Mul : PhysicalVariable.mk (a.value*b.value) (a.dim*b.dim)
```

This definition makes use of the multiplication defined for the value type and dimension. Therefore, an operator can be defined for a physical variable as long as the operator exists for both the value type and dimension. The other operators: division, powers, addition, and subtraction, are defined in the same way, making use of the definitions for dimension.

Even though we can write physical equations that ensure dimensional homogeneity, there is one more thing missing for scientific computation: units, which provide a scale for a measurement. Defining units requires defining what "1" is, since this is the reference used to build measurements. We will make use of the SI units definition, as this is the basis of scientific calibration. The definition of the SI base units from the International Bureau of Weights and Measures is outlined in Table 3.

Table 3: Definition of the seven base SI units as of 2019 [32].

SI Unit	Definition
Second (s)	Defined as $9,192,631,770$ oscillations of the unperturbed ground-state hyperfine transition frequency of the Caesium (Cesium)-133 atom.
Meter (m)	Defined as setting the speed of light in vacuum to be $299,792,458$ in units of m/s, using the definition of second.
Kilogram (kg)	Defined as setting Planck's constant to be $6.62607015 \times 10^{-34}$ in units of kg m^2/s, using the definition of meter and second.
Ampere (A)	Defined as setting the elementary charge to be $1.602176634 \times 10^{-19}$ in units of A s, using the definition of the second.
Kelvin (K)	Defined as setting the Boltzmann constant to be 1.380649×10^{-23} in units of kg m^2/(s^{2}K), using the definition of kilogram, meter, and second.
Mole (mol)	Defined as $6.02214076 \times 10^{23}$ elementary entities.
Candella (cd)	Defined as setting the luminous efficacy of monochromatic radiation at a frequency of 540 THz to be 683 in units of cd sr/(kg m^{2}s^3), using the definition of kilogram, meter, and second.

The base dimensions are implemented in Lean below. Starting with time, the duration of the ground state hyperfine oscillation of Caesium-133 is defined as one time. From there, a second is defined as 9,192,631,770 of those oscillations.

```
/-- Definition of the unit of time for a single caesium-133 oscillation -/        (source)
def caesium133GroundStateHyperfineOscillationDuration (B : Type u) (V : Type v) [Field V]
    [HasBaseTime B] :
  PhysicalVariable B V := ⟨1, dimension.time B V⟩
/-- Definition of the second based on the caesium-133 atom -/
def second (B : Type u) (V : Type v) [Field V] [HasBaseTime B] :
  PhysicalVariable B V := 9192631770·caesium133GroundStateHyperfineOscillationDuration B V
```

The meter is defined as one length. With the definition of the meter and the second, we can then define the speed of light as exactly 299,792,458 m/s.

```
/-- Definition of the meter -/                                                     (source)
def meter (B : Type u) (V : Type v) [Field V] [HasBaseLength B] :
  PhysicalVariable B V := ⟨1, dimension.length B V⟩
/-- Definition of the speed of light using the meter and second -/
def SpeedofLight [HasBaseLength B] [HasBaseTime B] : PhysicalVariable B V := 299792458 · meter B
    V/second B V
```

Following the same theme, the remaining five base units are defined as one of their respective dimension and the values of the five corresponding physical constants are fixed with those units.

```
/-- Definition of the kilogram -/                                       (source)
def kilogram (B : Type u) (V : Type v) [Field V] [HasBaseMass B] :
  PhysicalVariable B V := ⟨1, dimension.mass B V⟩
/-- Definition of the ampere -/
def ampere (B : Type u) (V : Type v) [Field V] [HasBaseCurrent B] :
  PhysicalVariable B V := ⟨1, dimension.current B V⟩
/-- Definition of the kelvin -/
def kelvin (B : Type u) (V : Type v) [Field V] [HasBaseTemperature B] :
  PhysicalVariable B V := ⟨1, dimension.temperature B V⟩
/-- Definition of the mole -/
def mole (B : Type u) (V : Type v) [Field V] [HasBaseAmount B] :
  PhysicalVariable B V := ⟨1, dimension.amount B V⟩
/-- Definition of the candela -/
def candela (B : Type u) (V : Type v) [Field V] [HasBaseLuminosity B] :
  PhysicalVariable B V := ⟨1, dimension.luminosity B V⟩
```

```
/-- Definition of Planck's constant using the meter, kilogram, and second -/   (source)
def PlancksConstant (B : Type u) (V : Type v) [Field V] [HasBaseLength B] [HasBaseTime B]
    [HasBaseMass B] [Pow V V] [SMul Float V]:
  PhysicalVariable B V := 6.62607015e-34·(kilogram B V * (meter B V)^2 / second B V)
/-- Definition of the Elementary Charge using the ampere and second -/
def ElementaryCharge (B : Type u) (V : Type v) [Field V] [HasBaseCurrent B] [HasBaseTime B]
    [SMul Float V]:
  PhysicalVariable B V := 1.602176634e-19 · (ampere B V * second B V)
/-- Definition of Boltzman's constant using the meter, second, and kelvin -/
def BoltzmannConstant (B : Type u) (V : Type v) [Field V] [HasBaseLength B] [HasBaseTime B]
    [HasBaseMass B] [HasBaseTemperature B] [Pow V V] [SMul Float V] :
  PhysicalVariable B V := 1.380649e-23 · (kilogram B V * (meter B V)^2 / ((second B V)^2 *
    kelvin B V))
/-- Definition of Avogadros Number from the mole -/
def AvogadrosNumber (B : Type u) (V : Type v) [Field V] [HasBaseAmount B] [Pow V V] [SMul Float
    V]:
  PhysicalVariable B V := 6.02214076e23 · (mole B V)^(-1)
/-- Defnition of the luminous efficacy of 540 THz monochromatic light -/
def MonochromaticRadiation540THz (B : Type u) (V : Type v) [Field V] [Pow V V]  [HasBaseLength
    B] [HasBaseTime B] [HasBaseMass B] [HasBaseLuminosity B] :
  PhysicalVariable B V := 683 · (candela B V* steradian B V / (kilogram B V * (meter  B V)^2 *
    (second B V)^3)
```

5   Discussion: Application to Scientific Computing with the Lennard Jones Potential

Ugwuanyi et. al. showed how Lean could be used to create a formalized and executable framework to compute molecular interaction energies with the Lennard-Jones potential [25]. However, they did this by using real numbers for their proofs and floats for their executions, without incorporating units into the Lean code. In this section, we show how this code can incorporate our physical variable definition to allow us to also ensure dimensional and unit homogeneity. To do so, we will show how the Lennard-Jones potential can be defined in Lean with physical variables and two theorems about the Lennard-Jones potential.

The Lennard-Jones potential is a model to describe the potential energy between two molecules, represented by Eq. 8, where V is the potential energy, ε is the depth of the potential well (the minimum energy in the interaction), σ is the distance where the energy between the two molecules is zero (attractive and repulsive forces are balanced), and r is the distance between the molecules.

$$V = 4\varepsilon \left(\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^6 \right) \tag{8}$$

In Lean, we define this as the following:

```
/-- Structure of Lennard-Jones parameters -/                          (source)
structure LJparams (B V) [Field V] [HasBaseLength B] [HasBaseTime B] [HasBaseMass B] where
  (σ ε : PhysicalVariable B V)
  (dimσ : σ.dim = dimension.length B V)
  (dimε : ε.dim = dimension.energy B V)
/-- Definition of the Lennard-Jones potential -/
noncomputable def LJ {B V} [Field V] [HasBaseLength B] [HasBaseTime B] [HasBaseMass B] (p :
    LJparams B V) (r : PhysicalVariable B V):
  PhysicalVariable B V := 4 · (p.ε * ((p.σ/r)^(12) - (p.σ/r)^6))
```

The first part is a structure that describes the parameters of the Lennard-Jones potential. An example of the implementation of this structure is shown below using the HCB LJ-1S model for hydrogen [33]. It contains the two parameters and the requirements for the dimensions of each parameter. The second is the definition of the Lennard-Jones potential, which takes in the parameter structure and the distance between the two molecules. One thing to note is that the definition does not put any requirement on the dimensions of LJ or r. One could add a premise to the definition requiring r to have units of length, but it would be an unused premise. This is a semantic issue where, while the user knows a priori that the Lennard-Jones potential describes energy, there is no formal reason to constrain the dimension of r. Therefore, it would not be incorrect to pass a temperature as r in LJ, as it is simply an equation involving three physical variables. However, we would encounter an error if we attempted to pass the temperature as r when using LJ in a proof to describe energy. We will see in the final theorem that requires a constraint on r because we are talking about energy.

```
/-- Example application of the Lennard-Jones parameter structure for hydrogen -/
/-- From the HCB LJ-1S model by Hirschfelder, Curtiss, and Bird -/
instance hydrogenLJ (B V) [Field V] [HasBaseLength B] [HasBaseTime B] [HasBaseMass B] [SMul
    Float V]: LJparams B V where
  σ := 0.2928·(nanometer B V)
  ε := 5.1097e-22·(joule B V)
  dimσ := rfl
  dimε := rfl
```

The first theorem we show is that the Lennard-Jones potential gives zero energy when the radius equals σ. Now that we are using the Lennard-Jones potential to describe energy and we will need r to have the dimension of length. From our Lennard-Jones structure, σ is required to have dimensions of length, and this property is used since σ is substituted in for r.

```
/-- Theorem proving that when the molecules are separated by distance σ, the energy    (source)
is 0 -/
theorem LJ_zero_energy {B V} [Field V] [HasBaseLength B] [HasBaseTime B] [HasBaseMass B] (p :
    LJparams B V) (hσ : p.σ.value ≠ 0) :
LJ p p.σ = ⟨0,dimension.energy B V⟩ := by
/-- rest of proof on GitHub-/
```

The other theorem we will show is the derivative of the Lennard-Jones potential with respect to the distance between the molecules, which describes the force between the two molecules. This makes use of our definition of the single variable derivative for physical variables, which we go over in the supplementary information.

```
/-- Theorem showing the force between two molecules that follow a          (source)
Lennard-Jones potential -/
theorem LJ_deriv {B V} [NontriviallyNormedField V] [HasBaseLength B] [HasBaseTime B]
    [HasBaseMass B]  (p : LJparams B V)
(r : PhysicalVariable B V) (hr0 : r.value ≠ 0) (hr : r.dim = dimension.length B V) :
    PhysicalVariable.deriv (LJ p) r =  4· (p.ε * (-12·p.σ^12/r^13 + 6·p.σ^6/r^7)) := by
/-- rest of proof on GitHub-/
```

6 Conclusion

Here, we have shown how dimensional analysis and physical variables can be defined using Lean. An interactive theorem prover was chosen over other program options because of its consistency and normalizing characteristics. We began with an overview of physical variables and dimensions, focusing on their mathematical properties. Then we showed how to define dimensions in Lean and showed that they form an Abelian group. We showed how to write theorems about dimensional homogeneity of equations and the implementation of the Buckingham Pi Theorem. Then, we showed how to use our definition of dimensions to create physical variables and implemented units based on the definitions from the International Bureau of Weights and Measures. Finally, we showed an application of this code, building off the Lennard-Jones Lean code developed by Ugwunayi et. al. This highlighted the ability to ensure dimensional consistency within proofs and the ease of application of the code. This code should provide the framework necessary to construct physically meaningful equations and perform scientific computations in Lean.

Acknowledgements

References

(1)   Fourier, J., *Théorie analytique de la chaleur*, 1822.

(2)   Çengel, Y. A.; Cimbala, J. M., *Fluid mechanics: fundamentals and applications*, Third edition; McGraw Hill: New York, 2014; 1000 pp.

(3)   Gehani, N. *Computer Languages* **1977**, *2*, 93–111.

(4)   McBride, C.; Nordvall-Forsberg, F. In *Advanced Mathematical and Computational Tools in Metrology and Testing XII*; Series on Advances in Mathematics for Applied Sciences Volume 90, Vol. Volume 90; WORLD SCIENTIFIC: 2021, pp 331–345.

(5)   Bennich-Björkman, O.; McKeever, S. In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering*, Association for Computing Machinery: New York, NY, USA, 2018, pp 121–132.

(6)   Gehani,   N.   H.   *Software: Practice   and   Experience*   **1985**,   *15*,   _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380150604, 555–569.

(7)   Cmelik, R.; Gehani, N. *IEEE Software* **1988**, *5*, Conference Name: IEEE Software, 21–27.

(8)   Kennedy, A. J. *Programming languages and dimensions*; UCAM-CL-TR-391; University of Cambridge, Computer Laboratory, 1996.

(9)   Gundry, A. *ACM SIGPLAN Notices* **2015**, *50*, 11–22.

(10)   Karam, M.; Saad, T. *SoftwareX* **2021**, *16*, DOI: 10.1016/j.softx.2021.100851.

(11)   Kennedy, A. In *Central European Functional Programming School: Third Summer School, CEFP 2009, Budapest, Hungary, May 21-23, 2009 and Komárno, Slovakia, May 25-30, 2009, Revised Selected Lectures*, Horváth, Z., Plasmeijer, R., Zsók, V., Eds.; Springer: Berlin, Heidelberg, 2010, pp 268–305.

(12)   Owre, S.; Saha, I.; Shankar, N. In *FM 2012: Formal Methods*, ed. by Giannakopoulou, D.; Méry, D., Springer: Berlin, Heidelberg, 2012, pp 356–371.

(13)   Muradian, R.; Urintsev, A. L. *Mathcmatica in Education* **1994**, *E2-94-110*, 1–24.

(14)   Albrecht, M. C.; Nachtsheim, C. J.; Albrecht, T. A.; Cook, R. D. *Technometrics* **2013**, *55*, Publisher: ASA Website _eprint: https://doi.org/10.1080/00401706.2012.746207, 257–270.

(15) Hutter, K.; Jöhnk, K. In *Continuum Methods of Physical Modeling: Continuum Mechanics, Dimensional Analysis, Turbulence*, Hutter, K., Jöhnk, K., Eds.; Springer: Berlin, Heidelberg, 2004, pp 339–392.

(16) Tao, T. A mathematical formalisation of dimensional analysis, What's new `https://terrytao.wordpress.com/2012/12/29/a-mathematical-formalisation-of-dimensional-analysis/`.

(17) Collins, J. B. In *Intelligent Computer Mathematics*, Autexier, S., Campbell, J., Rubio, J., Sorge, V., Suzuki, M., Wiedijk, F., Eds., Series Title: Lecture Notes in Computer Science; Springer Berlin Heidelberg: Berlin, Heidelberg, 2008; Vol. 5144, pp 370–381.

(18) Sonin, A. A., *The Physical Basis of dimension analysis*, 2nd, 2001.

(19) De Moura, L.; Ullrich, S. In Automated Deduction – CADE 28, 2021, pp 625–635.

(20) Paulson, L. C. *The Journal of Logic Programming* **1986**, *3*, 237–258.

(21) Norrish, M.; Slind, K. *The Computer Journal* **2002**, *45*, 37–45.

(22) Bobbin, M. P.; Sharlin, S.; Feyzishendi, P.; Dang, A. H.; Wraback, C. M.; Josephson, T. R. *Digital Discovery* **2024**, *3*, Publisher: RSC, 264–280.

(23) Tooby-Smith, J. *Computer Physics Communications* **2025**, *308*, Publisher: Elsevier, 109457.

(24) Tooby-Smith, J. Formalization of physics index notation in Lean 4, 2024.

(25) Ugwuanyi, E. D.; Jones, C. T.; Velkey, J.; Josephson, T. R. Benchmarking Energy Calculations Using Formal Proofs, 2025.

(26) Boer, J. d. *Metrologia* **1995**, *31*, 405.

(27) Newell, D. B.; Tiesinga, E. *The international system of units (SI): 2019 edition*; NIST SP 330-2019; Gaithersburg, MD: National Institute of Standards and Technology, 2019, NIST SP 330–2019.

(28) Technical Committee ISO/TC 12, I. O. f. S. In 2nd; Part 1: General Part 1, Vol. ISO 80000-1:2022; International Organization for Standardization: 2022.

(29) Männer, R. *SIGPLAN Not.* **1986**, *21*, 11–20.

(30) Bell, J. L. *Journal of Philosophical Logic* **1993**, *22*, Publisher: Springer, 1–18.

(31) Wirth, C.-P. *Journal of Applied Logic* **2008**, *6*, 287–317.

(32) *The International System of Units (SI)*; 9th edition; International Bureau of Weights and Measures: 2019.

(33) Hirschfelder, J. O.; Curtiss, C. F.; Bird, R. B., *Molecular theory of gases and liquids.* John Wiley & Sons Ltd: 1954.

FORMALIZING DIMENSIONAL ANALYSIS USING THE LEAN THEOREM PROVER
SUPPLEMENTARY INFORMATION

Maxwell P. Bobbin[1], Colin Jones[1], John Velkey[1], and Tyler R. Josephson[1,2]

[1]Department of Chemical, Biochemical, and Environmental Engineering, University of Maryland Baltimore County, 1000 Hilltop Circle, Baltimore, MD 21250
[2]Department of Computer Science and Electrical Engineering, University of Maryland Baltimore County, 1000 Hilltop Circle, Baltimore, MD 21250

7 The Buckingham π Theorem

The Buckingham π Theorem states that, given a set of dimensions that form a dimensional matrix, we can calculate the number of dimensionless numbers that can be formed and construct those numbers. The dimensionless matrix of a list of n dimensions with k base dimensions is a $k \times n$ matrix where each entry, (i, j), corresponds to the value of the exponent for base dimension i in variable j. For the set of dimensions: (length, time, and velocity), the dimensional matrix would look like:

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & -1 \end{bmatrix} \tag{9}$$

Since the dimensional matrix must be of a form where the rows correspond to base dimensions and the columns correspond to the variables, we create a definition for the dimensional matrix in Lean so we can ensure all dimensional matrices are of the same form.

```
/--Converts a list (tuple) of dimensions (the variables) into a matrix          (source)
of exponent values-/
def dimensional_matrix {n : ℕ}  [Fintype B] (d : Fin n → dimension B E) (perm : Fin
    (Fintype.card B) → B) : Matrix (Fin (Fintype.card B)) (Fin n) E := Matrix.of.toFun (fun (a
    : Fin (Fintype.card B)) (i : Fin n) => d i (perm a))
```

This definition requires two fields: the list of variables, which has the type *Fin n → dimension α γ* meaning a tuple of n dimensions, and the permutation of the base dimensions. The permutation is not unique and is a way of picking a numerical order in which the base dimensions are indexed. By taking in a permutation, we can write a traditional matrix, like the one shown above. A possible permutation for *SpatialTemporalSystem*, defined in a previous section, is:

```
/-- Example permutation definition using the SpatialTemporal system -/
def SpatialTemporalSystemPerm
| (0 : Fin 2) => SpatialTemporalSystem.Length2
| (1 : Fin 2) => SpatialTemporalSystem.Time2
```

The two parts of the Buckingham π Theorem determine how many dimensionless numbers, called Pi groups, can be formed and what those dimensionless numbers are. The number of pi groups that are possible is given by Equation 10, where n is the number of parameters and k is the rank of the matrix. The rank of the matrix represents how many unique base dimensions describe the variables.

$$p = n - k \tag{10}$$

In Lean, this is defined as:

```
/--Defnition of the number of dimensionless parameters possible from a     (source)
list of dimensions-/
noncomputable def number_of_dimensionless_parameters {n : ℕ}  [Fintype B] (d : Fin n →
    dimension B E) (perm : Fin (Fintype.card B) → B) := n - Matrix.rank (dimensional_matrix d
    perm)
```

The rank of the dimensional matrix will normally be equal to the cardinality of the system in use. However, there are cases where this won't be true, specifically if a system contains a base dimension that isn't used. For example, if we consider finding the Pi groups for the set of variables: length and area, using *system1*, the matrix will look like:

$$
\begin{bmatrix} 1 & 2 \\ 0 & 0 \end{bmatrix}
\tag{11}
$$

Finding the form of the dimensionless parameters is done by finding the kernel of the dimensional matrix.

```
/--Defnition of the dimensionless parameters from a list of dimensions (not unique)-/     (source)
def dimensionless_numbers_matrix {n : ℕ}  [Fintype B] (d : Fin n → dimension B E) (perm : Fin
    (Fintype.card B) → B) := LinearMap.ker (Matrix.toLin' (dimensional_matrix d perm))
```

8 Derivatives of Physical Variables

Derivatives are ubiquitous in engineering calculations and, here, we show the implementation of derivatives on a single variable in Lean. Like most operators on physical variables, the derivative operates on the value and the dimension separately. For a single variable function, the definition of the derivative is:

$$
\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}
\tag{12}
$$

Looking at the dimensions of this equation, we recognize that h must have the same dimension as x since h is added to x. Then, $f(x+h)$ must have the same dimension as $f(x)$. Therefore, we can simplify this equation and find that the dimension of the derivative is (recognizing that the limit does not change the dimension of the formula):

$$
\frac{df(x)}{dx} [=] \frac{f(x)}{x}
\tag{13}
$$

When it comes to dimensions, the derivative acts just like division.

```
/-- Definition of the derivative and integral operator for a single          (source)
variable dimension function -/
def derivative (f : dimension B E → dimension B E) (b : dimension B E) : dimension B E := (f
    b)/b
def integral (f : dimension B E → dimension B E) (b : dimension B E) : dimension B E := (f b)*b
```

Then, for a physical variable function, the derivative is defined as:

```
/--Definition of the derivative for a single physical variable function -/     (source)
protected def deriv {B : Type u} {V : Type v} [NontriviallyNormedField V] (f : PhysicalVariable
    B V → PhysicalVariable B V) (x : PhysicalVariable B V) : PhysicalVariable B V :=
let val' := deriv (PhysicalVariable.to_val_fun f) x.value
let dim' := dimension.derivative (PhysicalVariable.to_dim_fun f) x.dim
⟨val', dim'⟩
```

This has two parts: the derivative of the value, which uses the derivative definition from Mathlib, and the derivative of the dimension, which uses our definition of the derivative for dimensions. It also uses two definitions which converts a physical variable function into a value function or a dimension function:

```
/-- Converts a physical variable function into a function of the value -/          (source)
protected def to_val_fun {B : Type u} {V : Type v} [Field V] (f : PhysicalVariable B V →
    PhysicalVariable B V) : V → V
| a => (f ⟨a,dimension.dimensionless B V⟩).value
/-- Converts a physical variable function into a function of the dimensions -/
protected def to_dim_fun {B : Type u} {V : Type v} [Field V] (f : PhysicalVariable B V →
    PhysicalVariable B V) : (dimension B V) → (dimension B V)
| d => (f ⟨0,d⟩).dim
```