BACHELOR

Waterproof

Educational Proof Assistant for Linear Algebra

Bastiaansen, Bram

Award date:
2023

Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science

Waterproof: Educational Proof Assistant for Linear Algebra

Bachelor End Project

B. Bastiaansen

Supervisors:
J.W. Portegies
I. Heckenberger

Eindhoven, July 2023

Abstract

In an effort to help teach students in the writing of mathematical proofs, the educational software *Waterproof*[1], based on the proof assistant *Coq*[2], was developed at Eindhoven University of Technology (TU/e). Waterproof has been used as a teaching aid in the Analysis 1 course at the TU/e for several years, and has generally received positive reviews from students. In this report we explore the possibility of using Waterproof in a similar manner to supplement teaching of the course Linear Algebra at the Philipps-Universität Marburg, Germany. To achieve this, we adapt and add to the Coq library *coq-waterproof*[3], which is an extension (both in terms of theory and tactics) on the default Coq proof assistant. Throughout this report we include many examples and code snippets to demonstrate and explain the use of the Waterproof in the mathematical setting of Linear Algebra. We make significant developments in the formalization of algebraic theory in Waterproof, and gather sufficient evidence to conclude that Waterproof could be used in such an educational setting, bringing the assimilation of programmable proofs and their hand-written equivalents one step closer.

[1] *Waterproof* (version 0.6.1) can be accessed via `https://github.com/impermeable/waterproof`
[2] *Coq* is a formal proof management system, accessible via `https://coq.inria.fr`
[3] *coq-waterproof* (version (1.2.4) can be accessed via `https://github.com/impermeable/coq-waterproof`

Contents

Chapter 1

Introduction

As artificial intelligence (AI) permeates throughout more and more aspects of life, we believe that it is likely that AI will start to play a more active role in the mathematical community. In particular, the automation of mathematical proof verification and its use in an educational setting has picked up in popularity in recently ([Wiedijk, 2003], [Dénès et al., 2012], [Wemmenhove et al., 2022]). The ideas and work presented in this report aim to contribute to the development and advancement of the use of AI in what we consider to be the foundation of mathematics; discovering, proving and teaching mathematics.

First-year mathematics students often report struggling with the writing of abstract and theoretical mathematical proofs. In an effort to help teach students at Eindhoven University of Technology (TU/e) to formulate a proper mathematical proof, TU/e professor J.W. Portegies and his team developed an educational software called Waterproof [Wemmenhove et al., 2022]. This education tool is based on type theory and the interactive theorem prover known as Coq ([Coquand and Huet, 1985], [Bertot and Pierre, 2004], [Inria, 1995]). In this program, students can type out their proofs in a style that is meant to closely resemble conventional hand-written notations. While students do this, Waterproof's automation system checks the logical soundness of every step and provides immediate feedback to students. For several years now, Waterproof has been used as a supplementary learning aid in teaching the course Analysis 1 at the TU/e.

I. Heckenberger, a professor from the University of Marburg (Philipps-Universität Marburg), reached out to the TU/e inquiring after the possibility of using Waterproof to supplement teaching of the first-year course Linear Algebra in a similar fashion to which it has been done for Analysis 1 at the TU/e. In this report, we explore this prospect by constructing a custom library of formalized algebraic theory in Waterproof with the aim to assist in setting up an educational module for students taking Linear Algebra at the University of Marburg. This line of work builds on the efforts produced by team ChefCoq ([Wemmenhove et al., 2022], [Beurskens, 2019]), but specializes in algebraic concepts, aiming to lay foundations for Linear Algebra in Waterproof that may (in the future) be used as a complementary or alternative way for students to study the course. The idea is therefore to start doing for Linear Algebra what Waterproof has done for Analysis 1.

The main purpose of this report is to investigate whether Waterproof is able to help students in learning how to write proofs for Linear Algebra. What this means in more concrete terms is the following:

1. We consider a select few topics from the lecture notes provided for Linear Algebra ([Heckenberger, 2023]) that feature some interesting types of proofs and algebraic theory.
2. If we can cover the corresponding material in Waterproof, then we do so by constructing the necessary structure and support. In this case, we also show-case through examples how

Waterproof might assist students in their understanding of the course and more generally, their understanding of mathematical proofs.

3. If Waterproof is not readily able to cover (some) theoretical aspects of the course, then we explore and try to execute the steps necessary to equip Waterproof to handle these aspects.

While the ultimate goal is to help students taking Linear Algebra through the interactive education software Waterproof, the priority of this report is to establish (first) whether or not Waterproof is an appropriate tool for the job, and (second) to create coding infrastructure and support for any future educational modules in Waterproof. Finally, since covering the entirety of the course Linear Algebra is too grand a task, we instead focus on a select few specific areas.

1.1 Linear Algebra

As an ancient and well-defined area of mathematics (and a required first-year course for many mathematical degrees), we assume that readers have some fundamental understanding of the study of vectors, dimensions, and other introductory algebraic concepts. In this report we base our definitions and conventions of this discipline on the lecture notes provided by I. Heckenberger [Heckenberger, 2023] (adapted slightly in some aspects to suit our specific programming needs). Students are expected to work with several introductory algebraic concepts such as fields, vector spaces, and linear combinations of vectors. Furthermore, students taking Linear Algebra at the University of Marburg should be able to use the theory covered in classes to prove simple statements such as the example shown below:

Lemma 3.10. For an arbitrary field (X) and an element $x \in X$ in that field,

$$\begin{aligned} &(1) \text{ if } \forall a \in X, \quad x + a = a, \quad \text{then } x = 0. \\ &(2) \text{ if } \forall a \in X, \quad x \cdot a = a, \quad \text{then } x = 1. \end{aligned}$$

Such a proof is relatively simple using the axioms of a field (more on this later in Section 3.2), and might be written on pen-and-paper as follows:

Proof of (1):
Let $x \in X$ and take $a \in X$. Assume that $x + a = a$. We need to show that $x = 0$.
We conclude that $x = x + 0 = x + (a - a) = (x + a) - a = a - a = 0$.

For such a short proof, it is important that students learn how to structure their proofs efficiently, logically and correctly. This is precisely where Waterproof may offer assistance, since it enforces proper structure and has students write out their proofs mechanically in a way that resembles the writing of code, while simultaneously sticking as close as possible to the handwritten version of the proof. We will cover the proof of this lemma in Waterproof in Chapter 3.2.2.

Later on in the course, students look at more complicated statements, and eventually work up to proving a statement such as the one shown below:

Lemma 4.26. Let $n \in \mathbb{N}$, let $\vec{b}_1, \ldots, \vec{b}_n \in V$ (vectors in a vector space), let $\lambda_1, \ldots, \lambda_n \in X$ (scalars in a field), and let $\vec{v} = \sum_{k=1}^{n} \lambda_k \cdot \vec{b}_k$ (a linear combination). If $(\vec{b}_1, \ldots, \vec{b}_n)$ is a **basis** of V, then for every $1 \leq i \leq n$ with $\lambda_i \neq 0$,

$$B_i = (\vec{b}_1, \ldots, \vec{b}_{i-1}, \vec{v}, \vec{b}_{i+1}, \ldots, \vec{b}_n)$$

is also a **basis** of V.

Proving such a statement involves notions of linear combinations, linear independence, generating systems of a vector space and more advanced algebraic theory. We will investigate the possibility of incorporating these algebraic concepts in Waterproof.

1.2 Proof Assistants

Created in the attempt to assist and improve the process of proof writing, special computer programs and software called proof assistants such as Coq have been used by teachers and researchers alike to study mathematics. In these systems, users can set up mathematical theories, define properties and perform logical reasoning. Using the particular syntax and notation of these programs, users can type out their proofs and have the system automatically verify the correctness of every step. Thanks to their ability to provide immediate and direct feedback to users, proof assistants have recently been used by teachers to reinforce and teach 'correct' proof writing ([Wemmenhove et al., 2022], [Bartzia et al., 2022]). Unfortunately, many proof assistants have a high entry barrier in the form of a steep learning curve. This turns out to be challenging for many students [Aldrich et al., 2008], and generally the use of proof assistants is mostly restricted to individuals who specialize in mathematical theories and type theory [Geuvers, 2009].

The earliest examples of proof assistants can arguably be traced back to conception of the term 'artificial intelligence'. In fact, the very first proof assistant may have been the Logic Theorist produced in 1956 ([Gugerty, 2006], [Newell and Simon, 1956]), considered by many to the be the first artificial intelligence program. Another early system for the automatic verification of proofs was Automath, developed by de Bruijn ([de Bruijn, 1994], [TU/e, 2003]), who incidentally was a professor of mathematics at Eindhoven University of Technology. This formalized language for the expression of complete mathematical theorems in computer programs was based on sound mathematical principles. In particular, the Curry-Howard Isomorphism [Poernomo et al., 2007] and the De Bruijn criterion established that in complete logical systems such as Coq, users can treat proofs themselves as mathematical objects. For more information on the mathematics behind these proof assistants including Coq, see ([Bastiaansen, 2023]). While the pioneering systems such as Automath laid the foundation for automated theorem proving, modern proof assistants have evolved significantly in terms of their representation, capabilities and interactivity.

1.2.1 Coq Proof Assistant

Since the internal environment of Waterproof is based in Coq, we devote some attention here to elaborate on this originally French proof assistant named after a rooster (see Figure 1.1). In simple terms, Coq is an interactive theorem prover. What that entails is the following: users can formalize mathematical concepts in Coq and have the system assist them in interactively writing mechanically-checked proofs. In some sense, Coq can be thought of as a programming language with some highly specialized features. Throughout history, Coq has been used to formally verify many important mathematical results, such as the famous four color theorem [Gonthier, 2005]. For an overview of other well-known theorems formalized in Coq, see ([Wiedijk, 2023]). Coq's core language ([Huet, 1992]) features many built-in tactics, and further allows for the option to create highly customize-able new tactics, definitions, notations, etc. In addition, Coq features a expansive variety of libraries supported and updated continuously by a large community, including the Mathematical Components library [Mahboubi and Tassi, 2021] and various other libraries covering algebraic constructs.

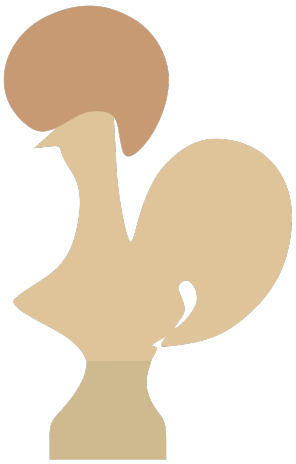

Figure 1.1: Coq logo

Coq itself is rather difficult to learn, and we will not spend significant effort in this report to try to explain this proof assistant's syntax. For a quick comprehensive tutorial on the use of Coq, see ([Bertot, 2006] [Barras et al., 1997]). This report mainly focused on investigating the application of the *coq-waterproof* library to the specific mathematical context of Linear Algebra. In particular, we construct the basic building blocks of Linear Algebra from the ground up, and incorporate elements from existing Coq libraries such as the Mathematical Components library to expand the *coq-waterproof* library.

1.2.2 Issues with Proof Assistants

Despite many attempts to lower the adoption barrier (learning curve) and align typed proofs with hand-written ones, there are a number of reported issues that remain. Some of the most prevalent issues include the following [Wemmenhove et al., 2022]:

- Learning the syntax and notation of a specific proof assistant can prove to be challenging, especially so for first-year mathematics students who might not yet feel comfortable with tactics and phrases used in ordinary pen-and-paper proofs.

- While many proof assistant like Coq allow for Unicode [1] notation, they do not support LaTeX[2]-formatted expressions. The Unicode Standard simply does not compare to the versatility and expressiveness of LaTeX-formatted expressions, which are much more closely aligned with what mathematical writing looks like on paper.

- The ability to write proofs in a proof assistant does not necessarily translate to an improved ability to write proofs with pen-and-paper. One study by Aldritch, Simmons and Shin [Aldrich et al., 2008] suggests that students who used the proof assistant SASyLF performed slightly better (on average) on an exam than students who did not. On the other hand, a study of a small computer science course [Knobelsdorf et al., 2017] using Coq IDE indicated that students performed worse at writing proofs with pen-and-paper than with Coq.

1.3 Waterproof

As part of an effort to address the issues listed above, team ChefCoq and their supervisor J.W. Portegies developed Waterproof [Wemmenhove et al., 2022], a custom software based on the Coq proof assistant. Designed to make a proof assistant more accessible for new students, the writing of a proof in Waterproof closely resembles the writing of a proof with pen-and-paper, albeit it in a slightly more mechanically structured way. See Figure 1.2 below for an example of Waterproof's interface.

[1] The Unicode Standard is the information technology standard for the consistent encoding, representation, and handling of text.

[2] LaTeXis a high-quality typesetting system, and is the standard for the communication and publication of scientific documents: `https://www.latex-project.org`

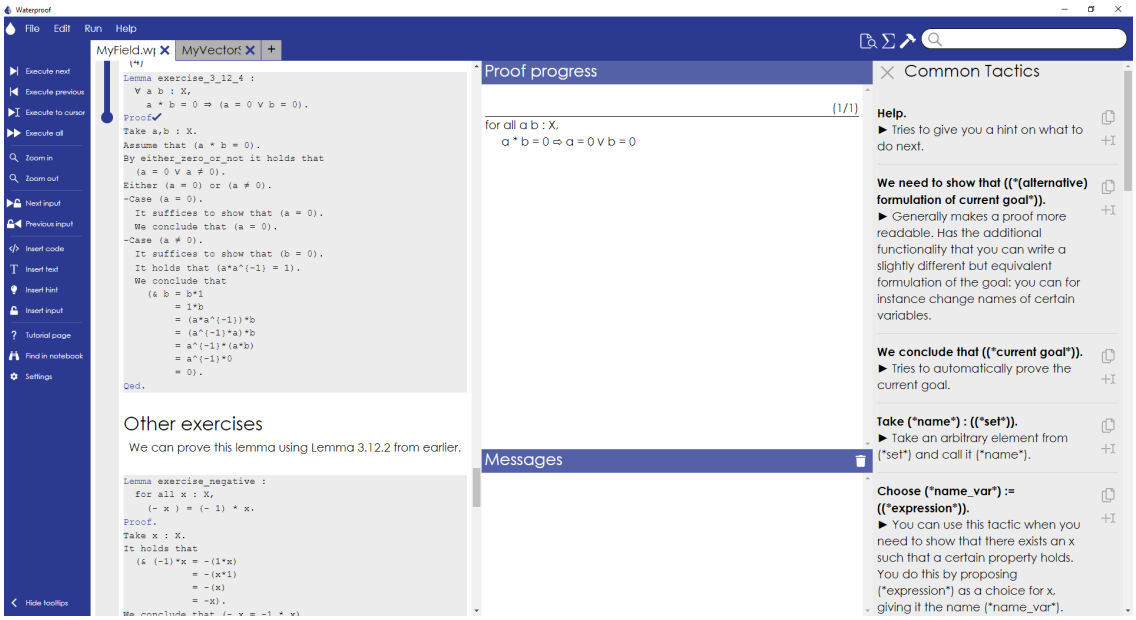

Figure 1.2: Screenshot of Waterproof. The right-hand side features a hide-able side-panel with an overview of coq-waterproof tactics, together with explanations.

Waterproof consists of a custom Coq library *coq-waterproof* and an editor designed to cater to the needs of teachers and students. The library *coq-waterproof* alters the style of Coq proofs to one more similar to pen-and-paper proofs while the editor allows for an easy-to-use interface with LaTeX-formatted expressions. Waterproof offers a detailed tutorial that is simple and intuitive to follow, allowing students to use standard proof tactics while keeping their proofs readable and properly structured. Since Waterproof is based on the proof assistant Coq, it inherits all of its syntax, notations and tactics, which can all be tailored to suit personal preferences.

1.4 Report Overview

In Chapter 2 we discuss and compare alternative methods and proof assistants that have been implemented in an educational setting, particularly in the mathematical discipline of linear algebra. In Chapter 3 we cover the introductory concepts of fields and vector spaces, and their initialization in Waterproof. Furthermore, in this chapter we provide various examples of proofs and exercises that can readily be applied as supplementary (or complementary) learning material for the course Linear Algebra. Chapter 4 covers the construction of support for algebraic concepts that Waterproof is not (yet) equipped for, which includes concepts such as linear independence, bases of vector spaces, etc. Finally, Chapter 5 details a discussion and review of the progress made in this report, with a conclusion that can be found in Chapter 6.

A significant part of this report is dedicated to demonstrating the use-case of Waterproof as a learning aid in the form of evidence via proofs. In particular, Chapters 3 and 4 account for the largest portion of the report, and both feature many subsections detailing topics (with corresponding examples of proofs) that build on one another, growing more complex as the report progresses. An executive summary of the report can be found in Appendix A. Full access to the code in this report can be found in this Github repository.

With this report we hope to build evidence and excitement for the use-case of Waterproof in a wider range of mathematical subjects, assisting new students with the writing of formal mathematical proofs, specifically those taking Linear Algebra at the University of Marburg.

Chapter 2

Related work

Proof assistants offer the potential to revolutionize the way we approach mathematical proofs and formal reasoning. However, due to their inherent complexity and steep learning curve, the adoption of proof assistants in an educational setting has seen limited use ([Geuvers, 2009], [Aldrich et al., 2008]). This has led to several attempts focused on the development of custom proof assistants that offer a more user-friendly environment, with an emphasis on their application in teaching mathematical proof writing techniques.

One such effort is the development of Waterproof by team ChefCoq [Wemmenhove et al., 2022]. In this paper we operate within the internal environment of Waterproof (i.e. its underlying structure) and thus benefit from the extensive library of tactics, notations and support that has been developed over the years. Since the field of research focused on the development educational and user-friendly proof assistants is rather broad, we limit ourselves to acknowledging the efforts made by others to help establish the formalization and teaching of Linear Algebra in proof assistants.

For instance, the team responsible for the development of the Lean Forward project has been working on the formalization of the entire undergraduate mathematics curriculum (including of course Linear Algebra) through the use of the Lean proof assistant [de Moura et al., 2015]. As stated on the Lean Forward website [1], the 'ultimate aim is to develop a proof assistant that actually helps mathematicians, by making them more productive and more confident in their results.' As this is a large-scale project that has not yet achieved said aim, we cannot give a conclusive statement about the effectiveness of this project. Instead, we merely reflect on its potential to enhance mathematical understanding and promote the use of proof assistants in the class-room setting.

Another such effort is C-Corn, short for the Constructive Coq Repository at Nijmegen, a rich mathematical library of constructive algebra and analysis formalized in the proof assistant Coq [Cruz-Filipe et al., 2004]. Constructive mathematics is the philosophical viewpoint that insists on the necessity to have constructive proofs (being able to provide concrete examples), as opposed to classical mathematics which allows for the existence of mathematical objects without concrete examples. As described by the authors, the aim of C-Corn was to establish a library of formalized mathematics that is accessible, readable, coherent and extensible. In other words, to create a 'computer system in which a mathematician can do mathematics'. While this is not a very beginner-friendly tool, it has contributed significantly to to the field of constructive algebra, which is part of Linear Algebra.

Another example of an earlier attempt to teach Algebra in an interactive way was the

[1] Accessible via `https://lean-forward.github.io`

Algebra Interactive project [Cohen et al., 1999] by professors from Eindhoven University of Technology in 1999. Covering first and second year Algebra, this project turned a basic course in Algebra into an interactive course using Java applets to present algebraic algorithms. While this is not the same as using a modern proof assistant like Coq in an educational setting, it demonstrates that the idea and use of computers in assisting the teaching of Linear Algebra has been around for a while.

Finally, we note that within the proof assistant Coq, there are several libraries which present their own formalization of algebraic constructs. For example, the Coq Standard Library [2] is composed of numerous general-purpose libraries with definitions and theorems for vectors, finite types, etc. Several of these libraries contain a rich source of formalized algebraic mathematics. Another example includes the Mathematical Components library [Mahboubi and Tassi, 2021] *mathcomp*[3], which covers a wide range of mathematical theory including algebraic topology such as vector spaces.

While many proof assistants are not yet specifically designed for educational purposes, the developing of interactive theorem proving environments, creation of educational tutorials, and the organization of workshops and seminars aimed at training teachers in the use of proof assistants has contributed in the efforts to make them more accessible and useful in an educational context. In particular, the development of user-friendly proof assistants like Waterproof in combination with the ongoing efforts to formalize subjects like Linear Algebra, are advancing what we believe to be a new era in the education of mathematics.

[2]The Coq Standard Library is available via `https://coq.inria.fr/library/index.html`

[3]The Mathematical Components library can be accessed via `https://math-comp.github.io/htmldoc/libgraph.html`

Chapter 3

Fields and Vector Spaces in Waterproof

3.1 Using Waterproof

With the initial goal of helping teach students to write mathematical proofs in the context of Analysis 1, Waterproof is tailor-made to accommodate the study of metric spaces, convergence and approximate behavior of functions. Since we would like to use the *coq-waterproof* library for the course Linear Algebra, we will need to incorporate the mathematical notations and conventions of algebraic theory. In particular, we begin by building the notion of a field from the ground up.

Before we start typing any lines of code into Waterproof, we need to import the tactics, notations and lemmas that have already been defined in the coq-waterproof library. We can do so by running the following lines of code.

```
Require Import Waterproof.AllTactics.
Require Import Waterproof.notations.notations.
Require Import Waterproof.load.
Module Import db_algebra := databases(Algebra).
```

3.2 Fields

The notion of a field (or *Körper* in German) refers to a set of elements on which addition and multiplication are defined, and all elements satisfy the corresponding field axioms (see the definition below). There are multiple equivalent definitions of a field (i.e. a rational domain such as the the real numbers $\mathbb{R}$ or the the complex numbers $\mathbb{C}$), and the specific inference rules or axioms can vary slightly in different definitions. As mentioned earlier, we base our working definitions and axioms on the material provided by the course lecture notes [Heckenberger, 2023]. Specifically, the lecture notes offer the following definition of a field:

Definition: Let X be a set with at least two elements, and let the zero element (0) and the unit element (1) be elements of X with $1 \neq 0$. Let addition and multiplication be two operations defined as follows:

$$+ : X \times X \to X,\ (a, b) \mapsto a + b, \quad * : X \times X \to X,\ (a, b) \mapsto a * b.$$

For abbreviation, one often writes ab or $a \cdot b$ instead of $a * b$. We call X (together with these choices of $0, 1, +, *$) a field with zero 0 and unit element 1 if the following applies:

(1) $+$ and $*$ are commutative:

$$\forall a, b \in X: \quad a + b = b + a \quad \wedge \quad a * b = b * a$$

(2) $+$ and $*$ are associative:

$$\forall a, b, c \in X: \quad (a + b) + c = a + (b + c) \quad \wedge \quad (a * b) * c = a * (b * c)$$

(3) $*$ is distributive:

$$\forall a, b, c \in X: \quad a * (b + c) = a * b + a * c$$

(4) 0 and 1 are neutral elements for $+$ and $*$, respectively:

$$\forall a \in X: \quad 0 + a = a \quad \wedge \quad 1 * a = a$$

(5) Existence of an additive inverse:

$$\forall a \in X, \exists b \in X: \quad a + b = 0$$

(we write $b = -a$; we write expressions of the form $a + (-b)$ with $a, b \in X$ as $a - b$.)

(6) Existence of a multiplicative inverse:

$$\forall a \in X \backslash \{0\}, \exists b \in X: \quad a * b = 1$$

(we write $b = a^{-1}$ or $b = \frac{1}{a}$.)

If one omits the last rule (existence of multiplicative inverses) in this definition, one speaks of commutative rings (such as the natural numbers $\mathbb{N}$) instead of a field. We make a note here that in the definition that we use in Waterproof, we deviate slightly from the choice of axioms outlined above. In particular, for axioms (5) and (6) we omit the existence of an inverse element b and instead introduce the inverse functions $-a$ and a^{-1} for addition and multiplication, respectively. This helps to make the notation more concise and allows us to write our proof in Waterproof in a more efficient manner. In other words, we use the slightly altered axioms:

(5') Existence of an additive inverse: $\forall a \in X: \quad a + (-a) = 0$

(6') Existence of a multiplicative inverse: $\forall a \in X \backslash \{0\}: \quad a * a^{-1} = 1$

3.2.1 Fields in Waterproof

Thanks to the relatively simple structure of a field and Coq's convenient way to define sets, operations and notations locally, we are able to construct a field in Waterproof completely from scratch. In this report we do not cover every instance of Coq syntax, and refer to ([Barras et al., 1997]) for any technical details. We begin by declaring all of the required initial parameters and functions (i.e. the zero and unit elements). See the code snippet below.

```
Parameter X : Type.              (* Field X *)

Parameter plus : X -> X -> X.    (* Addition *)
Parameter mult : X -> X -> X.    (* Multiplication *)
Parameter zero : X.              (* 0 *)
Parameter one : X.               (* 1 *)
```

```
Parameter opp : X -> X.          (* Subtraction *)
Parameter inv : X -> X.          (* Multiplicative inverse *)
```

Next, we declare a custom scope, which we call *field_scope*, allowing us to assert custom notation for specific elements and operators in our field. Doing this ensures that the automation system correctly parses our inputs; for instance it should be able to recognize that by typing 0 we refer to zero as an element of our field, and not as the element 0 of the natural numbers.

In the code snippet below, the line 'at level 35' essentially indicates the level of importance of the notation, where the level of importance increases as the level decreases. In other words, the automation system evaluates expressions with lower levels before expressions with higher levels. The specific choice of 35 is more or less arbitrary, but chosen such that it leaves enough space below and above it (the level can range from 1 to 200).

```
Declare Scope field_scope.

Infix "+" := plus : field_scope.
Infix "*" := mult : field_scope.
Notation "1" := one : field_scope.
Notation "0" := zero : field_scope.
Notation "- x" := (opp x) : field_scope.
Notation "- 1" := (opp one) : field_scope.
Notation "x ^{-1}" := (inv x) (at level 35) : field_scope.
Infix "-" := (fun a b : X => plus a (opp b)) : field_scope.

Delimit Scope field_scope with field.
Open Scope field_scope.
```

Most of the notation defined above is self-explanatory, and while we will not elaborate on every instance of notation that we introduce, we will briefly describe the subtraction $(-)$ function in the code snippet above. Whereas addition is a function that inherently belongs to the definition of a field, subtraction is not and can in some sense be viewed as specific case of addition. For example, we can write

$$0 - 1 \quad \text{as} \quad 0 + (-1).$$

Defining subtraction as follows ensures that we are still working under the specific set of results and axioms that define and govern our mathematical objects.

```
Infix "-" := (fun a b : X => plus a (opp b)) : field_scope.
```

Having defined this standard syntax that closely resembles what we associate with the notation on paper, we can declare our field axioms based on the aforementioned definitions. Note that this syntax is in the form of the Unicode Standard, and could possibly be improved in the future to allow for LaTeX-formatted expressions. Using these field axioms below, we should now be able to prove any simple statement that follows directly from this set of rules that govern the behaviour of elements in our field.

```
Parameter commutative_addition :
  forall a b : X, a + b = b + a.

Parameter commutative_multiplication :
  forall a b : X, a * b = b * a.

Parameter associative_addition :
  forall a b c : X, (a + b) + c = a + (b + c).

Parameter associative_multiplication :
  forall a b c : X, (a * b) * c = a * (b * c).

Parameter distributivity :
  forall a b c : X, a * (b + c) = a * b + a * c.

Parameter one_is_neutral :
  forall a : X, a * 1 = a.

Parameter zero_is_neutral :
  forall a : X,  a + 0 = a.

Parameter additive_inverse  :
  forall a : X, a + (-a) = 0.

Parameter multiplicative_inverse :
  forall a : X\{0}, a * a^{-1} = 1.
```

Figure 3.1: Field Axioms

3.2.2 Proofs with Fields in Waterproof

Now that we have defined the field axioms, we can use them to formally prove some simple algebraic properties. In other words, we can begin gathering evidence of Waterproof's use-case for Linear Algebra in an educational setting. In the code snippet below we include a proof of the uniqueness of the additive inverse.

In every line of structured logical reasoning, we explicitly mention the appropriate field axiom that we need to use to conclude the corresponding result. Notice further that the & symbol appears in chains of equalities of the form ($\&\_ = \_ = \_$). This is a by-product of one of Waterproof special features, which allows the functional use of these chains of equalities, something that Coq does not. While this is perhaps not a conventional method, it does help to identify the explicit use of these chains, forcing users to acknowledge the use of the equality chains.

```
Lemma uniqueness_additive_inverse :
  forall a b c : X,
    a + b = 0 -> a + c = 0 -> b = c.
Proof.
Take a, b, c : X.
Assume that (a + b = 0).
Assume that (a + c = 0).
```

```
By zero_is_neutral it holds that
  (& b = b + 0 = b + (a + c)).
By associative_addition it holds that
  (b + (a + c) = (b + a) + c).
By commutative_addition it holds that
  (& (b + a) + c = (a + b) + c = 0 + c).
By commutative_addition it holds that
  (0 + c = c + 0).
By zero_is_neutral it holds that
  (c + 0 = c).
We conclude that (b = c).
Qed.
```

Although a proof with pen-and-paper might be shorter than this, the sentence structure and flow of the proof closely resembles the ideal of hand-written proofs that teachers might want students to aspire to.

We include another example, showing that we can prove lemmas from the lecture notes of the course Linear Algebra [Heckenberger, 2023]. In particular, we prove the aforementioned Lemma 3.10 here (see Section 1.1), which consists of two results; the first of which states that if the sum of two elements a and x is equal to a, then x must be the zero element.

```
Lemma exercise_3_10_1:
  forall x : X, forall a : X, x + a = a -> x = 0.
Proof.
Take x,a : X.
Assume that (x + a = a) (i).
We need to show that (x = 0).
By zero_is_neutral it holds that (x = x + 0).
By additive_inverse it holds that
  (x + 0 = x + (a + (-a))).
By associative_addition it holds that
  (x + (a - a ) = (x + a) - a).
By (i) it holds that
  ((x + a) - a = a - a).
By additive_inverse it holds that
  (a - a = 0).
We conclude that (x = 0).
Qed.
```

In a classroom setting, students are generally not expected to explicitly state which theorem/axiom they are using in every single line of the proof, as this is a time-consuming process and makes the writing of proofs even more difficult for new students. Instead, students usually work under the assumption that the results that follow directly from inference rules such as the field axioms can be used implicitly in any proof writing. We can offer this option in Waterproof as well, by resolving prior results as hints in the *wp_algebra* database. This is done by executing the following line of code:

```
#[export] Hint Resolve 'some_result' : wp_algebra.
```

For instance, if we wanted to allow students to use the aforementioned Lemma 3.10.1 implicitly in future work, then we would execute the following line in Waterproof:

```
#[export] Hint Resolve lemma_3_10_1 : wp_algebra.
```

Now that we have the option to load our field axioms directly into the library, we may henceforth use them implicitly in our proofs.

If we were to take another look at Lemma 3.10.1 from above, then we can now use the resolved field axioms implicitly to skip some steps. Doing this allows us to immediately conclude the proof, which more closely aligns with what a hand-written proof might look like. See the code snippet below.

```
Lemma exercise_3_10_1_simplified:
  forall x : X, forall a : X, x + a = a -> x = 0.
Proof.
Take x, a : X.
Assume that (x + a = a).
We need to show that (x = 0).
We conclude that
  (& x = x + 0
       = x + (a - a)
       = x + a - a
       = a - a
       = 0).
Qed.
```

Likewise, the second part of Lemma 3.10; which states that if the product $xa = a$ for all $a \in X$ (where X is our field), then x must be the unit element, becomes easy to prove.

```
Lemma exercise_3_10_2:
  forall x : X, (forall a : X, x * a = a) -> x = 1.
Proof.
Take x : X.
Assume that (for all a : X, x * a = a) (i).
By (i) we conclude that
  (& x = x * 1 = 1).
Qed.
```

With every statement or lemma that we prove in Waterproof, we have more tools at our disposal, more blocks to build with. To see more examples of these types of proofs (and more), see the file *MyField.v*. We should make note here that Waterproof (version 0.6.1) had some trouble evaluating classical logic, such as the following statement: 'Either an element $a = 0$ or $a \neq 0$'. Consequently, as a by-product of the problems that we ran into, new features were incorporated into Waterproof to allow for full functionality of these statements of classical logic.

Reflection

With the introduction of a little notation, some imported syntax, and lemmas for support, we find that Waterproof is perfectly capable of acting as a proof builder and automatic checker for simple algebraic statements in a field. We provide sufficient examples in the file *MyField.v* and in Section 3.2.2 above to submit as evidence towards this case. We conclude that Waterproof can readily be used as a supplementary or complementary method of study for this isolated, introductory part of the course Linear Algebra.

3.3 Vector Spaces

The notion of a vector space (or *Vektorraum* in German) refers to a set of elements which is closed under addition and scalar multiplication. A common example includes $\mathbb{R}^n$, the n-dimensional Euclidean space, where every element (vector) can be interpreted as a list of n real numbers. A vector space (V) has two operations: addition is the component-wise addition of the elements of two vectors, and scalar multiplication is the individual multiplication of every element of a vector by a scalar (an element of a field X). In particular, we define a vector space V based on the lecture notes as follows [Heckenberger, 2023]:

Definition: Let V be a non-empty set with at least two elements. Let vector addition and scalar multiplication be the two operations defined as follows:

$$+ : V \times V \to V,\ (\vec{v}, \vec{w}) \mapsto \vec{v} + \vec{w}, \quad * : X \times V \to V,\ (\lambda, \vec{v}) \mapsto \lambda * \vec{v}.$$

We call V a **vector space** if the following conditions hold:

(1) $+$ is commutative:

$$\forall \vec{v}, \vec{w} \in V : \quad \vec{v} + \vec{w} = \vec{w} + \vec{v}$$

(2) $+$ is associative:

$$\forall \vec{u}, \vec{v}, \vec{w} \in V : \quad (\vec{u} + \vec{v}) + \vec{w} = \vec{u} + (\vec{v} + \vec{w})$$

(3) $*$ is associative:

$$\forall \lambda, \mu \in X,\ \forall \vec{v} \in V : \quad \lambda * (\mu * \vec{v}) = (\lambda * \mu) * \vec{v}$$

(4) Distributive laws:

$$\forall \lambda, \mu \in X,\ \forall \vec{v}, \vec{w} \in V : \quad (\lambda + \mu) * \vec{v} = \lambda * \vec{v} + \mu * \vec{v} \quad \wedge \quad \lambda * (\vec{v} + \vec{w}) = \lambda * \vec{v} + \lambda * \vec{w}$$

(5) Axiom of the zero (null) vector:

$$\forall \vec{v} \in V : \quad 0 * \vec{v} = \vec{0}$$

(we say $\vec{0}$ is the null vector, note that it is really different from the zero element 0 in a field)

(6) Scalar multiplication by 1:

$$\forall \vec{v} \in V : \quad 1 * \vec{v} = \vec{v}$$

3.3.1 Vector Spaces in Waterproof

Readers with a keen eye may notice that the notation for addition $(+)$ and multiplication $(*)$ is now overloaded, because we have identical notation for these operations in a field. Note however, that these are not the exactly same, addition in a field maps elements from a field (X) to that same field while addition in a vector space maps vectors to vectors. The elements in fields and vectors are different mathematical objects and should be treated as such.

Newer students sometimes find it difficult to identify when we are referring to addition or multiplication in a field as opposed to addition and scalar multiplication in a vector space. Furthermore, since we have already defined the '+' and '$*$' notation locally in the *field_scope* in Waterproof, we cannot re-use this notation in the scope of a vector space (doing so would require us to locally change the scope of the work-proof-environment every single time we wanted to perform a different operation, which is terribly inefficient). For these reasons, we introduce the following notation:

'+:' refers to vector addition, and '$*$:' refers to scalar multiplication.

Although this is not necessarily a conventional notation, we hope that typing proofs with this syntax allows students to better distinguish between the operators, and grasp the concepts faster. We use this new notation to build our notion of a vector space in Waterproof using the following lines of code.

```
Parameter V : Type.
Parameter plus_V : V -> V -> V.
Parameter scalar_mult_V : X -> V -> V.
Parameter one_V : V.
Parameter zero_V : V.

Declare Scope vector_space_scope.
Infix "+:" := plus_V (at level 42) : vector_space_scope.
Infix "*:" := scalar_mult_V (at level 40) : vector_space_scope.
Notation "0_V" := zero_V : vector_space_scope.
Notation "1_V" := one_V : vector_space_scope.
Notation "-: v" := (scalar_mult_V (opp one) v) (at level 42) :
    vector_space_scope.
Infix "-:" := (fun v w : V => plus_V v (scalar_mult_V (opp one) w))
    (at level 42) : vector_space_scope.

Delimit Scope vector_space_scope with vector_space.
Open Scope vector_space_scope.
```

Here, the following line of code assert that by subtracting one vector $\vec{w}$ from another vector $\vec{v}$, we are first using scalar multiplication of the (multiplicative) inverse of the field unit element 1 with $\vec{w}$ to obtain $-\vec{w}$, and then applying vector addition on the two vectors to obtain $\vec{v} + (-\vec{w}) = \vec{w}$. Enforcing this specific order ensures that the system parses this expression in the exact way we want it to.

```
Infix "-:" := (fun v w : V => plus_V v (scalar_mult_V (opp one) w))
```

Notice that similar to what we did earlier for fields, we have declared another custom scope here called *vector_space_scope*, so that we may introduce our new set of notation for vector spaces. The zero-vector in our vector space in Waterproof is denoted by "0_$V = 0_V$" ($\vec{0}$) in order to distinguish it from the zero element "0" in a field. The text 'at level 40/42' establishes the priority in which Waterproof evaluates computations. These levels were chosen to avoid clashes in notation priority when we introduce external libraries later in Section 4.2. With this, we can define our definition of a vector space in Waterproof.

```
Parameter commutative_addition_V :
  forall u v : V, u +: v = v +: u.

Parameter associative_addition_V :
  forall u v w : V, (u +: v) +: w = u +: (v +: w).

Parameter associative_scalar_multiplication_V :
  forall a b : X, forall v : V, a *: (b *: v) = (a * b) *: v.

Parameter distributivity_V_1 :
  forall a b : X, forall v : V,
    (a + b) *: v = a *: v +: b *: v.

Parameter distributivity_V_2 :
  forall a : X, forall v w : V,
    a *: (v +: w) = a *: v +: a *: w.

Parameter null_vector_V :
  forall v : V, 0 *: v = 0_V.
  (* Reads 0*v is equal to the zero vector. *)

Parameter one_is_neutral_V :
  forall v : V,  1 *: v = v.
  (* Here 1 is the unit element from a field. *)
```

Figure 3.2: Definition of Vector Space

3.3.2 Proofs with Vector Spaces in Waterproof

Now that we have defined how elements should behave in a vector space, we can prove some algebraic results from the course lecture notes (pages 40-42) [Heckenberger, 2023]. In particular, we start by proving that (vector) addition of an arbitrary vector with the zero vector is just the vector itself, a very intuitive and yet important result that we can directly derive from the definitions above in Figure 3.2.

Lemma 4.5

```
Lemma lemma_4_5_1 :
  forall v : V, 0_V +: v = v.
Proof.
Take v : V.
By one_is_neutral_V it holds that
  (1*:v = v).
By null_vector_V it holds that
  (& 0_V +: v = 0 *: v +: v
              = 0 *: v +: 1 *: v).
By distributivity_V_1 we conclude that
  (& 0_V +: v = 0 *: v +: 1 *: v
              = (0 + 1) *: v
```

```
                = (1 + 0) *: v
                = 1 *: v
                = v).
Qed.
```

Notice that here we explicitly use the defined laws of the vector space, and implicitly used the field axioms defined earlier in Figure 3.1. A proof with pen-and-paper would not use this explicit structure or any of the semi-colon notations that we've introduced here, but we argue that this notation might help students distinguish between operations in a field and operations in a vector space. We include the proof of the second statement of Lemma 4.5 as well, where the notation

```
v +: (-1) *: v = 0_V
```

in the code snippet below should read as "$\vec{v} - \vec{v} = \vec{0}$."

```
Lemma lemma_4_5_2 :
  forall v : V, v +: (-1)*: v = 0_V.
Proof.
Take v : V.
By one_is_neutral_V it holds that
  (1*:v = v).
It holds that
  (v +: (-1)*: v = 1 *: v +: (-1)*: v).
By distributivity_V_1 it holds that
  (& v +: (-1)*: v = 1 *: v +: (-1)*: v
                   = (1 + (-1)) *: v
                   = 0 *: v).
By null_vector_V we conclude that
  (& v +: (-1)*: v = 0 *: v
                   = 0_V).
Qed.
```

Once again, we note that in practice, students are not expected to quote every single property from which they derive a result. For something basic and essential like the definition of vector space, students should simply be able to implicitly use these results in their proof-writing. Similar to what we did in the field scope, we can resolve the axioms of a vector space (as well any other previous results we want) in the *wp_algebra* database to make our proof-writing more efficient. We will showcase some more examples of what these proofs might look like.

Theorem 4.6

This theorem from [Heckenberger, 2023] features 5 statements, which perfectly illustrate the use-case of Waterproof for helping to teach students how to write such proofs. The theorem goes as follows:

If X is a field and V is a vector space, then:

$$
\begin{aligned}
&(1) \quad \forall a \in X, \quad a \cdot \vec{0} = \vec{0} \\
&(2) \quad \forall a \in X, \forall \vec{v} \in V, \quad (-a) \cdot \vec{v} = -a \cdot \vec{v} = a \cdot (-\vec{v}) \\
&(3) \quad \forall a \in X, \forall \vec{v} \in V, \quad a \cdot \vec{v} = \vec{0} \implies (a = 0) \vee (\vec{v} = \vec{0}) \\
&(4) \quad \forall a \in X \backslash \{0\}, \forall \vec{v}, \vec{w} \in V, \quad a \cdot \vec{v} = a \cdot \vec{w} \implies \vec{v} = \vec{w} \\
&(5) \quad \forall a, b \in X, \forall \vec{v} \in V \backslash \{\vec{0}\}, \quad a \cdot \vec{v} = b \cdot \vec{v} \implies a = b
\end{aligned}
$$

We can prove all of these statements in Waterproof, in a way that closely resembles the pen-and-paper proof equivalents, but we will only showcase a couple here. For access to all the complete proofs, see the file *MyVectorSpace.v*.

Starting with an easier proof to get a sense of the structure, we include an example of the proof of statement (1) in the code below. This proof merely requires some creative use of rewriting using the axioms of both fields and vector spaces.

```
Theorem theorem_4_6_1 :
  forall a : X, a *: 0_V = 0_V.
Proof.
Take a : X.
We conclude that
  (& a *: 0_V = a *: (0 *: 0_V)
              = (a * 0) *: 0_V
              = 0 *: 0_V
              = 0_V).
Qed.
```

Note that the proof of statement (4) below is rather long, due to the necessary amount of rewriting steps to obtain the required results. Herein lies a possible current weakness of Waterproof as a proof checker engine; at times the system cannot process the correctness of a step if it deems that the step taken is too large. For instance, in the proof below the system would not accept the following conclusion straight away

$$(a^{-1} \cdot a) \cdot (\vec{v} - \vec{w}) = 1 \cdot (\vec{v} - \vec{w})$$

Instead, the system requires the statement $\left(a \cdot a^{-1} = a^{-1} \cdot a = 1\right)$ to be asserted before-hand in order to process the line above as correct. One possible work-around for this issue is to rigorously store all such statements in the database prior to writing any proofs. This approach however, present its own set of problems; for instance, care would need to be taken to ensure that students could not abuse the automation system by skipping too many steps. There is a very little compromise in the Waterproof checking-engine with regards to mathematical rigor, every step needs to be written out in sufficient detail, with the correct assumptions and results established a priori.

```
Theorem theorem_4_6_4 :
  forall a : X, forall v w : V,
    a is not equal to 0 -> (a *: v = a *: w) -> (v = w).
Proof.
Take a : X.
Take v, w : V.
Assume that (a is not equal to 0).
Assume that (a *: v = a *: w).
```

```
(* We use Theorem 4.6.2 here. *)
It holds that
  (& 0_V = a *: w -: a *: w
         = a *: v -: a *: w
         = a *: v +: a *: (-:w)
         = a *: (v -: w)).
It holds that
  (& a^{-1} * a = a * a^{-1} = 1).
(* With that out of the way, we show that. *)
It holds that
  (& 0_V = a^{-1} *: (0_V)
         = a^{-1} *: (a *: (v -: w))
         = (a^{-1} * a) *: (v -: w)
         = 1 *: (v -: w)
         = (v -: w)).
(* Now we add w to both sides and rewrite. *)
We conclude that
  (& v = 0_V +: v
       = v +: 0_V
       = v +: (w -: w)
       = v +: (-:w +: w)
       = (v -: w) +: w
       = 0_V +: w
       = w).
Qed.

#[export] Hint Resolve theorem_4_6_4 : wp_algebra.
```

Lastly, we showcase the proof of statement (5) in the code snippet below. Here we highlight one of our vexes with Coq's automation system. In the code below, we start by deriving that $\vec{0} = (a-b)\cdot\vec{v}$. We know that we can now use statement (3), which we had already proved before, to derive that either $a-b=0$ or $\vec{v}=\vec{0}$. This is a necessary step to continue our proof, but there is a slight issue here. Namely, since we programmed statement (3) to require an assumption of the form: $(a-b)\cdot\vec{v}=\vec{0}$, the system cannot parse that we have met this assumption. Even though we have proved that $\vec{0} = (a-b)\cdot\vec{v}$, this is not sufficient for the automation engine. Instead, we are required to first rewrite the result above using reflexivity of the equality sign before we may use statement (3).

While this is perhaps only a little inconvenience brought about by the strictness of Coq's notation system, it is possible that small inconveniences such as these can undermine students' trust in a program that advertises itself as a flexible proof checker. If we want Waterproof to more closely resemble hand-written proofs, care needs to be taken to address flexibility issues.

```
Theorem theorem_4_6_5 :
  forall a b : X, forall v : V\{0_V},
    (a *: v = b *: v) -> (a = b).
Proof.
Take a,b : X.
Take v : V.
Assume that (v is not equal to 0_V).
Assume that (a *: v = b *: v).
```

```
(* We subtract b*v from both sides *)
It holds that
  (& 0_V = b *: v -: (b *: v)
         = a *: v -: (b *: v)
         = a *: v +: (-b) *: v
         = (a - b) *: v).

(* We need to rewrite this into the exact form of Theorem 4.6.3. *)
It holds that
  ((a - b) *: v = 0_V).
(* Now we can use the theorem. *)
By theorem_4_6_3 it holds that
  ((a - b) = 0 OR v = 0_V) (i).

We claim that ((a - b) = 0).
We argue by contradiction.
Assume that ((a - b) is not equal 0).
(* By our assumption that v is not the zero vector *)
It holds that
  ((a - b) is not equal to 0 AND v is not equal to 0_V).
By (i) it holds that
  (negate ((a - b) is not equal to 0 AND v is not equal to 0_V)).
Contradiction.

(* Now that we shown that a - b = 0, the rest is easy. *)
We conclude that
  (& a = a + 0
       = a + (b - b)
       = a + (-b + b)
       = (a - b) + b
       = 0 + b
       = b + 0
       = b).
Qed.

#[export] Hint Resolve theorem_4_6_5 : wp_algebra.
```

Reflection

By adding the custom notation '+:' and '∗:' for operators in vector spaces, we combined and integrated the theory from fields with the theory of vector spaces. Our hope and intent is that this custom syntax serves to make proof-writing/typing in Waterproof more structured and mechanical, rather than confusing and intimidating for students. Whether or not this will prove to be the case will depend on students' personal experiences.

Once again, we showcased through examples in Section 3.3.2 that Waterproof works quite well as a proof assistant in this regard (for more examples see the file *MyVectorSpace.v*), and that there are many different methods and tactics that users can employ to derive results. With some small changes to the automation system (i.e. reducing the amount of cumbersome rewriting steps), Waterproof should be able to consistently help verify proofs in most of the introductory concepts of Linear Algebra.

Chapter 4

Expanding on Algebraic Theory in Waterproof

The notions of fields and vector spaces were relatively simple to construct from scratch, but now the topics and theory we would like to explore become a little more complicated in terms of the mathematics and type theory involved. In particular, we aim to prove some statements about vectors of finite dimensions, linear combinations of vectors, linear independence, and bases of vector spaces. Since there is not a large amount of literature or work dedicated to the formalization of these advances topics in Linear Algebra, working with the concepts in Waterproof requires us to build a substantial amount of infrastructure to support these formalized notions.

At this point in the report we deviate from catering to inexperienced Coq users with simple notation and code aimed to be user-friendly for first-year mathematics students, and instead delve into the jungle of formalized mathematical type theory that Coq has to offer. That is to say; up until this point we presented our results and reported on their successes and shortcomings. Now, we shift towards describing new results in a more general manner by reflecting on what we learned from them. The reason for this is because we simply produce too many results to adequately devote attention to all of them.

Although we will try to keep the code in a simple declarative style, we must provide a fair warning to those inexperienced with Coq's unique syntax. The code and the theory that we will discuss in the following sections are a little more advanced and sometimes may require some background knowledge to understand. In particular, we assume that readers have some understanding of bases, generators, linear independence, etc. For more background information on this topic, the book Linear Algebra (see [Nair and Singh, 2018]) is a good place to start.

To elaborate on the challenge we face here, consider the following. We have shown in Chapter 3 that we can readily prove (almost) any simple logical statement with elements in fields and vector spaces, and do so in a way such that the proofs in Waterproof closely resemble pen-and-paper proofs. Now we would like expand on these concepts and prove statements such as the following lemma from the lecture notes [Heckenberger, 2023]:

Lemma 4.26: Let $n \in \mathbb{N}$, let $\vec{b}_1, \ldots, \vec{b}_n \in V$, let $\lambda_1, \ldots, \lambda_n \in X$, and let $\vec{v} = \sum_{k=1}^{n} \lambda_k \cdot \vec{b}_k$. If $(\vec{b}_1, \ldots, \vec{b}_n)$ is a **basis** of V, then for every $1 \leq i \leq n$ with $\lambda_i \neq 0$, the tuple

$$B_i = (\vec{b}_1, \ldots, \vec{b}_{i-1}, \vec{v}, \vec{b}_{i+1}, \ldots, \vec{b}_n)$$

is also a **basis** of V.

In order to prove such a statement in Waterproof, we first need to establish formalizations

of some algebraic concepts and theory. In particular, we need to have some working definitions of linear independence, generator systems, bases, etc. Building all of these notions from scratch requires considerable expertise with dependent type theory and familiarity with the use of Coq, not to mention an exceedingly large amount of time to develop. Therefore, we instead look to what the mathematical community has to offer. This part of the report focuses on researching the different kinds of pre-existing infrastructure and support available for finite vectors. We try to establish our own formalization of this advanced algebraic theory.

4.1 The Coq Standard Library

The first option we investigate is the standard Coq library *Coq.Vectors*[1], which has a rendition of finite tuples of vectors $(\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n)$. For instance, in this library, the tuple $(\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n)$ of n vectors is defined as follows:

```
Notation: t V n = t(V(n)) = tuple of length n consisting of vectors in V.
```

After altering some notation and syntax, we establish the following way to define small vectors (see the code snippet below). This structure somewhat resembles the data-structure of lists commonly used in programming languages such as Python, and should feel familiar to most students following a technical degree. Furthermore, using some special non-dependent instantiations of induction schemes from the library *Coq.Vectors.VectorDef*, we managed to define some advanced vector operations such as the linear combinations of finite tuples of vectors. However, we did not manage to create very intuitive or user-friendly syntax for these conventions.

```
Parameter v1 v2 v3 : V.

Definition v := [ v1 , v2 , v3 ].
```

While this structure makes sense from a programming perspective, one might object that this notation is not very conventional for mathematicians. In order to introduce these concepts to students in Waterproof, the goal is to simplify everything to get as close to the hand-written conventions as possible. This programming with dependent types may instead serve to confuse students and should be avoided to reduce the learning curve of the proof assistant.

Reflection

Although we adapted the code from the Standard Coq library to suit our needs, we ultimately did not get very far in proving any interesting or useful properties using this particular library. This due to multiple factors, including the limited size and infrastructure developed in the *Coq.Vectors* library, as well as our own inexperience with Coq at this point in the report. Although this library has plenty of useful examples of formalized algebra, we could not make sufficient nor efficient progress under resource constraints, and instead opt to switch our focus to another library that offers alternative prospects.

4.2 Mathematical Components library

The second (and final) option we investigate is the extensive Mathematical Components library *mathcomp*[2]. This library consist of a plethora of formalized mathematics developed using the

[1] The Coq Standard Library is available via `https://coq.inria.fr/library/index.html`

[2] The Mathematical Components library can be accessed via `https://math-comp.github.io/htmldoc/libgraph.html`

Coq proof assistant [Mahboubi and Tassi, 2021], focusing on algebraic properties which we might be able to adapt and use in our work. In particular, the *mathcomp* library has entire branches dedicated solely to fields, vectors, finite types, and a sub-branch that presents the canonical 'big' operator (e.g. the Σ operator for summation) which we are most interested in.

4.2.1 Finite vectors

Within the Mathematical Components library, there is a library based on finite types, which features the construction of a set **ordinal n** (denoted by I_n) of integers $0 \leq i < n$, and functions over these finite types [Gonthier et al., 2007]. We can use this to define finite tuples of vectors as follows: consider a set of vectors $(\vec{v}_0, \vec{v}_1, \ldots, \vec{v}_{n-1})$ as a function from the finite set $[0, 1, \ldots, n-1]$ to (either) a field X (a tuple of scalars) or a vector space V (a tuple of vectors). For instance, the tuple of scalars $\vec{a} : I_n \mapsto X$ is a tuple of elements in the field X, with $a_i = a(i)$ as the $(i+1)$-th entry of the tuple. Similarly, the function $\vec{v} : I_n \mapsto V$ can be interpreted as a tuple of n vectors $(\vec{v}_0, \vec{v}_1, \ldots, \vec{v}_{n-1})$, where $\vec{v}_i = \vec{v}(i)$ is the $(i+1)$-th vector in this tuple.

4.2.2 Big Operators

Since we are looking to encapsulate all of the theory and practices associated with the topics such as bases and linear independence, one of the first crucial steps involves being able to handle 'big' summations such as

$$\sum_{i=0}^{n} \vec{v}_i.$$

The *bigop* sub-library from the Mathematical Components library features a design of such operators, and contains generic theory of such big operators, including many unique lemmas that perform complex operations such as re-indexing with minimal user input and minimal assumptions [Bertot et al., 2008]. In order to use these structures with the notions of a vector space (see Figure 3.2) we defined earlier, we simply need to meet the algebraic requirements on these spaces; which in this case are the following four common monoid laws:

1. Associativity of addition with vectors: $\forall \vec{x}, \vec{y}, \vec{z} \in V, \quad \vec{x} + (\vec{y} + \vec{z}) = (\vec{x} + \vec{y}) + \vec{z}$
2. Left addition of the zero element: $\forall \vec{x} \in V, \quad \vec{0} + \vec{x} = \vec{x}$
3. Right addition of the zero element: $\forall \vec{x} \in V, \quad \vec{x} + \vec{0} = \vec{x}$
4. Commutativity of addition with vectors: $\forall \vec{x}, \vec{y} \in V, \quad \vec{x} + \vec{y} = \vec{y} + \vec{x}$

Since we had already established such rules and results earlier in Section 3.3.2, it is very simple to prove that our library meets these conditions (for more details, see the file *MyFiniteVectors.v*). With these rules established, we can now access and use general notations of the form:

```
\big [ op / nil ]_ (index and range description ) F
```

Note that once again the notation adheres to the Unicode Standard, which encodes these mathematical expressions in way that somewhat closely resembles hand-written notation. For example, the big summation $\sum_{i=1}^{n} \vec{v}_i$ (equivalent to $\sum_{i=0}^{n-1} \vec{v}_i$) can now be typed in Waterproof as follows: provided that v is a function from $I_n \mapsto V$ (a tuple of n vectors), we type

```
\sum_(i < n) v(i).
```

Further integration of LaTeX-formatted expressions in Waterproof may help to display this notation in a manner that more closely resembles the hand-written equivalent.

4.2.3 Proofs with the MathComp library

The main advantage that the Mathematical Components library offers over the Coq Standard Library is the large number of lemmas that can readily be applied to the concepts and structure that we have built up in the previous sections. Using some clever rewriting tactics and dependent-type logic, we are able to obtain many useful lemmas from the *bigop* library [Bertot et al., 2008]. The list below includes a number of useful lemmas and statements that we were able to adopt after some efforts. Again, to view full details on these formalized proofs (and more results not mentioned in this list) in Waterproof, see the file *MyFiniteVectors.v*.

For all $n \in \mathbb{N}$, for all scalars $\lambda, \mu \in X$, and for all vectors $(\vec{v}_0, \vec{v}_1, \ldots, \vec{v}_{n-1}, \vec{v}_n) \in V$ and $(\vec{w}_0, \vec{w}_1, \ldots, \vec{w}_{n-1}, \vec{w}_n) \in V$:

(1) $\displaystyle\sum_{i=0}^{n} \vec{v}_i = \vec{v}_0 + \sum_{i=0}^{n-1} \vec{v}_{i+1}$

(2) $\displaystyle\sum_{i=0}^{n} \vec{v}_i = \sum_{i=0}^{n-1} \vec{v}_i + \vec{v}_n$

(3) $\displaystyle\sum_{i=0}^{0} \vec{v}_i = \vec{0}$

(4) $\displaystyle\sum_{i=0}^{n-1} \lambda \cdot \vec{v}_i = \lambda \cdot \sum_{i=0}^{n-1} \vec{v}_i$

(5) $\displaystyle\sum_{i=0}^{n-1} (\lambda \cdot \vec{v}_i + \mu \cdot \vec{w}_i) = \lambda \cdot \sum_{i=0}^{n-1} \vec{v}_i + \mu \cdot \sum_{i=0}^{n-1} \vec{w}_i$

(6) $\displaystyle\sum_{i=0}^{n-1} (\vec{v}_i + \vec{w}_i) = \sum_{i=0}^{n-1} \vec{v}_i + \sum_{i=0}^{n-1} \vec{w}_i$

(7) $\displaystyle\sum_{i=0}^{n-1} (\vec{v}_i + \vec{w}_i) = \sum_{i=0}^{n-1} \vec{v}_i + \sum_{i=0}^{n-1} \vec{w}_i$

(8) $\displaystyle\sum_{i=0}^{n-1} \vec{v}_i = \vec{v}_j + \sum_{\{i=0 \,|\, i \neq j\}}^{n-1} \vec{v}_i$

While these statements might seem rather straight-forward and intuitive, their formalization in Waterproof is not quite so trivial. In order to formalize any kind of mathematics in Coq, every single step must be logical and correct; the system must ensure that every step can be proven before another can be made. In particular, some of the proofs in the list above were rather complicated to establish in Waterproof.

For example, the proofs of (1) and (2) relied on other properties that were proven in yet another *mathcomp* library, such as a property that allows for interpreting an element $i \in I_n$ as an element of I_m if $n \leq m$, or another property that 'bumps' an element $i \in I_n$ to $i+1$ while ensuring that $i+1 \in I_{n+1}$. This strict dependence on maintaining rigorous type theory can quickly get out of hand and become difficult to manage. For instance, the proof of (4) required a proof by induction on n, using properties (2) and (3) combined with some rewriting steps conditioned over predicates.

Naturally we did not do all of this just for fun. All of these lemmas and statements that we collected serve a purpose, namely in that they are relevant in developing the concepts required to prove a statement such as **Lemma 4.26** mentioned earlier. For illustrative purposes, we will walk through yet another lemma that we need later in our proof of Lemma 4.26. This statement goes as follows:

Lemma: For all $n \in \mathbb{N}$, for all scalars $\lambda \in X$, for all sequences of scalars $(\mu_0, \mu_1, \ldots, \mu_{n-1}) \in X$, and for all vectors $(\vec{v}_0, \vec{v}_1, \ldots, \vec{v}_{n-1}, \vec{v}_n) \in V$:

$$\sum_{i=0}^{n-1} \lambda \cdot (\mu_i \cdot \vec{v}_i) = \sum_{i=0}^{n-1} (\lambda \cdot \mu_i) \cdot \vec{v}_i$$

We include the code snippet below:

```
Lemma sum_asociative_scalar_mult :
  for all n : N, for all L : X,
  for all F : 'I_n -> V, for all A : 'I_n -> X,
  for all P : 'I_n -> bool,
    \sum_(i < n | P i) L *: (A(i) *: F(i))
    = \sum_(i < n | P i) (L * A(i)) *: F(i).
Proof.
Take n : N. Take L : X.
Take F : ('I_n -> V). Take A : ('I_n -> X).
Take P : ('I_n -> bool).
By eq_bigr it suffices to show that
  (for all i : I_n, L *: (A(i) *: F(i)) = (L * A(i)) *: F(i)).
By associative_scalar_multiplication_V we conclude that
  (for all i : I_n, L *: (A(i) *: F(i)) = (L * A(i)) *: F(i)).
Qed.
```

Observe that even though the structure of the proof may feel somewhat 'mechanical', the style is readable and easy to follow. This proof in particular is a rather short example, mostly relying on yet another result *eq_bigr*, which essentially states the following:

$$\text{If for every index } i,\ \vec{v}_i = \vec{w}_i,\ \text{then it holds that } \sum_{i=0}^{n-1} \vec{v}_i = \sum_{i=0}^{n-1} \vec{w}_i.$$

Reflection

Whereas we could not make significant progress towards our goals in utilizing the Coq Standard Library, the opposite could be said for the Mathematical Components library. Thanks to the efforts of the Inria-Microsoft Research Join Center and the continuous upkeep by the Mathematical Components team, we were able to locate and successfully adapt many lemmas with greatly developed infrastructure for finite vectors and big summations, etc. The work done to establish these branches of formalized mathematics in Coq is not something we would have had time to develop, and hence expanding on the work by [Bertot et al., 2008] and others allows us to make remarkable progress in our mission to cover some more advanced theory of Linear Algebra in Waterproof.

4.3 Linear Algebra

After the success we experienced in working with the Mathematical Components library, we decided to continue our efforts based in this expansive collection of formalized algebra. It is now almost time to begin the proof of Lemma 4.26 which we have repeatedly alluded to. However, before we can discuss how we approached this proof, we must first formalize the notions of linear independence, bases, and other such concepts in Waterproof.

4.3.1 Proofs and Definitions

We have mentioned the concepts of linear independence, bases and generator systems and other such notions repeatedly throughout this paper while assuming that readers have a basic understanding of the terms. We continue to do so here, but we will specify exactly which definitions we use in the sections below. In these sections we will showcase how we used Coq's customize-able notation system to set up our own definitions of these concepts in Waterproof. For full access to the code in which we establish these definitions formally in Waterproof, see the file *MyLinearIndependence.v*.

Linear Independence

A tuple of vectors $(\vec{v}_0, \vec{v}_1, \ldots, \vec{v}_{n-1}) \in V$ is called linearly independent if for every sequence of scalars $(\lambda_0, \lambda_1, \ldots, \lambda_{n-1}) \in X$, the only solution to the equation

$$\sum_{i=0}^{n-1} \lambda_i \cdot \vec{v}_i = \vec{0}$$

is the trivial solution: so for every index i, it holds that $\lambda_i = 0$.

Equivalently in Waterproof,

```
Definition linearly_independent (n : N) (F : 'I_n -> V) :=
  for all A : 'I_n -> X,
    \sum_(i < n) (A(i) *: F(i)) = 0_V
      => (for all i : 'I_n, A(i) = 0).
```

Note that here the function $F : I_n \mapsto V$ represents the tuple of vectors $(\vec{v}_0, \vec{v}_1 \ \ldots, \vec{v}_{n-1})$.

Linear Dependence

A tuple of vectors $(\vec{v}_0, \vec{v}_1, \ldots, \vec{v}_{n-1}) \in V$ is called linearly dependent if they **not** linearly independent. An alternative definition we use is the following: A tuple of vectors $(\vec{v}_0, \vec{v}_1, \ldots, \vec{v}_{n-1}) \in V$ is called linearly dependent if there exists a non-trivial sequence of scalars $(\lambda_0, \lambda_1, \ldots, \lambda_{n-1}) \in X$ (so there exists at least one index i such that $\lambda_i \neq 0$) such that

$$\sum_{i=0}^{n-1} \lambda_i \cdot \vec{v}_i = \vec{0}.$$

Equivalently in Waterproof,

```
Definition linearly_dependent (n : N) (F : 'I_n -> V) :=
  ~ linearly_independent(n,F).
```

Generating System

A tuple of vectors $(\vec{B}_0, \vec{B}_1, \ldots, \vec{B}_{n-1}) \in V$ is called a generating system (or generator) of V if it holds that every vector $v \in V$ can be 'generated by' this generating system. In other words, for every $v \in V$, there exists a sequence of scalars $(\lambda_0, \lambda_1, \ldots, \lambda_{n-1}) \in X$ such that

$$v = \sum_{i=0}^{n-1} \lambda_i \cdot \vec{B}_i.$$

So v can be written as a linear combination of vectors $(\vec{B}_0, \vec{B}_1, \ldots, \vec{B}_{n-1})$. Note that the zero vector can be generated by any tuple of vectors by taking the trivial sequence of scalars.

Equivalently in Waterproof,

```
Definition generator_of_V (n : N) (B : 'I_n -> V) :=
  for all v : V, there exists A : 'I_n -> X,
    v = \sum_(i < n) (A(i) *: B(i)).
```

Bases

A tuple of vectors $(\vec{B}_0, \vec{B}_1, \ldots, \vec{B}_{n-1}) \in V$ is called a base (or basis) of V if they are **linearly independent** and it holds that every vector $v \in V$ can be written as a linear combination of vectors $(\vec{B}_0, \vec{B}_1, \ldots, \vec{B}_{n-1})$. In other words, for every $v \in V$, there exists a sequence of scalars $(\lambda_0, \lambda_1, \ldots, \lambda_{n-1}) \in X$ such that

$$v = \sum_{i=0}^{n-1} \lambda_i \cdot \vec{B}_i.$$

Analogously, we call the tuple of vectors $(\vec{B}_0, \vec{B}_1, \ldots, \vec{B}_{n-1})$ a base of V if the vectors are **linearly independent** and a form a **generating system** of V.

Equivalently in Waterproof,

```
Definition base_of_V (n : N) (B : 'I_n -> V) :=
  linearly_independent(n,B) AND
    (for all v : V, there exists A : 'I_n -> X,
      v = \sum_(i < n) (A(i) *: B(i))).
```

Proving Linearity Properties of Generators

In order to keep the paper concise, we will only go through the proof of one of the more important results we established using these definitions. In particular we showcase the following result:

Lemma: Suppose that we have two vectors $\vec{v}, \vec{w} \in V$ that are both generated by some tuple of vectors $(\vec{B}_0, \vec{B}_1, \ldots, \vec{B}_{n-1})$. Then if we were to take any two scalars $\alpha, \beta \in X$, it holds that also $(\alpha \cdot \vec{v} + \beta \cdot \vec{w})$ is generated by $(\vec{B}_0, \vec{B}_1, \ldots, \vec{B}_{n-1})$.

Utilizing the lemmas that we have collected in the previous section (4.2.3) in combination with our new set of definitions, this is not so difficult to prove in Waterproof. See the code snippets below. In the first code snippet we identify the specific goal that we need to achieve.

```
Lemma generated_by_linearity :
   for all n : N, for all B : 'I_n -> V, for all v w : V, for all a b : X,
    generated_by(n,v,B) => generated_by(n,w,B)
    => generated_by(n, (a*:v +: b*:w), B).
Proof.
Take n : N. Take B : ('I_n -> V).
Take v,w : V. Take a,b : X.
Assume that (generated_by(n, v, B)) (Hv).
Assume that (generated_by(n, w, B)) (Hw).
It suffices to show that
  (there exists A : 'I_n -> X,
     a *: v +: b *: w = \sum_(i < n) A(i) *: B(i)).
```

In the second code snippet, we obtain sequences of scalars $(\lambda_0, \lambda_1, \ldots, \lambda_{n-1})$, and $(\mu_0, \mu_1, \ldots, \mu_{n-1})$ such that we can write

$$v = \sum_{i=0}^{n-1} \lambda_i \cdot \vec{B}_i \text{ and } w = \sum_{i=0}^{n-1} \mu_i \cdot \vec{B}_i.$$

We then proceed to make an appropriate choice for the sequence of scalars $(A_0, A_1, \ldots, A_{n-1})$ by taking the linear combination of λ and μ with help of custom notation we defined for making such choices. In other words, for every index i, we have $A_i = \alpha \cdot \lambda_i + \beta \cdot \mu_i$. Note that in the code snippets, $a = \alpha, b = \beta, L = \lambda$, and $M = \mu$.

```
Obtain L according to (Hv), so for L : 'I_n -> X it holds that
  (v = \sum_(j < n) L(j) *: B(j)).
Obtain M according to (Hw), so for M : 'I_n -> X it holds that
  (w = \sum_(j < n) M(j) *: B(j)).
Choose A := (choose_gen_scalar(n,L,M,a,b)).
It suffices to show that
  (a *: (\sum_(j < n) L(j) *: B(j))
  +: b *: (\sum_(j < n) M(j) *: B(j))
  = \sum_(i < n) A(i) *: B(i)).
```

Using distributive properties of vector addition in sums and simplifying the goal, we are left with some simple rewriting steps after which the conclusion of the proof becomes rather easy. To view this proof in its entirety, see the file *MyLinearIndependence.v*. This file also features some additional proofs of several other interesting properties from the lecture notes up to the first two statements of Lemma 4.20. We would have liked to complete even more proofs, but since we were working under time constraints, and because we do not this particular lemma to derive later results, we did not complete all of Lemma 4.20. Instead, we skipped the completion of this proof in order to start working towards more complicated and interesting algebraic results. Nonetheless, we feel confident that the (partially) unfinished proof of Lemma 4.20 and other could be made complete with a little more time and effort.

4.3.2 Lemma 4.26

We are now finally equipped to tackle a more complicated algebraic result. For convenience, we restate the lemma here:

Lemma 4.26. Let $n \in \mathbb{N}$, let $\vec{b}_1, \ldots, \vec{b}_n \in V$ (vectors in a vector space), let $\lambda_1, \ldots, \lambda_n \in X$ (scalars in a field), and let $\vec{v} = \sum_{k=1}^{n} \lambda_k \cdot \vec{b}_k$ (a linear combination). If $(\vec{b}_1, \ldots, \vec{b}_n)$ is a **basis** of V, then for every $1 \leq i \leq n$ with $\lambda_i \neq 0$,

$$B_i = (\vec{b}_1, \ldots, \vec{b}_{i-1}, v, \vec{b}_{i+1}, \ldots, \vec{b}_n)$$

is also a **basis** of V.

Since we are working with a automation system that demands a certain standard of mathematical rigor, we need a way to define B_i as the tuple of vectors $(\vec{b}_1, \ldots, \vec{b}_n)$ with the $i-$th index replaced by the vector $\vec{v}$. If we were using pen-and-paper, then this would be trivial as we could simply define B_i as such using words. To get a computer system to recognize what we mean though, we need to properly establish a working definition. Luckily, Coq offers a way to create such custom definitions, as we demonstrate in the code snippet below. The definition *replace_index* does exactly what it sounds like: it replaces a vector with a given index in a tuple of vectors by any other vector we specify.

```
Definition replace_index (n : N) (F: 'I_n -> V) (j : 'I_n) (v : V) (i : 'I_n)
           :=  if (i == j) then v else F(i).
```

Using this definition, we can formally define B_i in Waterproof as follows:

```
replace_index(n,B,i,v).
```

Now since Lemma 4.26 wants us to prove that B_i is a basis of V, we ultimately need to show two things: that B_i is a linearly independent set of vectors, and that B_i is a generator of V. Since we can prove both of these things on their own, we split Lemma 4.26 into two slightly smaller lemmas, both of which we again break down into three main results each, all of which in turn can be further reduced to individual lemmas. Reducing the statement in this way helps manage the complexity of the overall procedure, breaking down the final proof into a series of smaller and simpler statements that will be combined in the end to construct a complete proof. In total, the complete proof of Lemma 4.26 consists of hundreds of lines of code, which is why we do not include the full code snippets here. For the full proof, see the file *Lemma_4_26.v*.

First Half of the Proof of Lemma 4.26

For this first half of the proof of Lemma 4.26, we need to show that B_i is a linearly independent set of vectors. Using the definitions established earlier in Section 4.3.1, this statement reduces to showing that for any sequence of scalars $\mu_1, \dots, \mu_n \in X$, the only solution to the equation

$$\sum_{j=0}^{n-1} \mu_j \cdot B_i(j) = \vec{0}$$

is the trivial solution: so for every index j, it holds that $\mu_j = 0$. Note that here $B_i(j)$ denotes the $j-$th vector in the tuple of vectors B_i, such that $B_i(i) = \vec{v}$.

Handwritten Proof Version

If we were to use pen-and-paper to write this proof, we might write something like this:

Proof: Assume that for any sequence of scalars $\mu_1, \dots, \mu_n$ it holds that $\sum_{j=0}^{n-1} \mu_j \cdot B_i(j) = \vec{0}$. Take an arbitrary index $j \in \{1, \dots, n\}$. Then it holds that

$$\begin{aligned}
\sum_{j=n}^{n} \mu_j \cdot B_i(j) &= \sum_{\{j=1 \,|\, j \neq i\}}^{n} \mu_j \cdot B_i(j) + \mu_i \cdot B_i(i) \\
&= \sum_{\{j=1 \,|\, j \neq i\}}^{n} \mu_j \cdot \vec{b}_j + \mu_i \cdot \vec{v} \\
&= \sum_{\{j=1 \,|\, j \neq i\}}^{n} \mu_j \cdot \vec{b}_j + \mu_i \cdot \sum_{k=1}^{n} \lambda_k \cdot \vec{b}_k \\
&= \sum_{\{j=1 \,|\, j \neq i\}}^{n} (\mu_j + \mu_i \cdot \lambda_j) \cdot \vec{b}_j + \mu_i \cdot \lambda_i \cdot \vec{b}_i \\
&= \vec{0}
\end{aligned}$$

Now using the fact that $\lambda_i \neq 0$, it follows from linear independence of the vectors $(\vec{b}_1, \dots, \vec{b}_n)$ that $\mu_i = 0$, and then also for all $j \in \{1, \dots, n\}$ it holds that $\mu_j = 0$. This is exactly what we needed to show, so we conclude that B_i is a linearly independent set of vectors.

Proof in Waterproof

While the proof using pen-and-paper only took a couple of lines, the proof in Waterproof is considerably more lengthy. In particular, this proof features more than 150 lines of code, and so we do not include the code snippets here since they would take up too much space! There are multiple reasons why this proof took so many lines, some of which we outline below:

1. Whereas in the handwritten version of the proof we make intuitive and correct steps by repeatedly rewriting and simplifying the sum, taking these corresponding steps in Waterproof requires more rigour. For example, the step

$$\sum_{\{j=1\,|\,j\neq i\}}^{n} \mu_j \cdot B_i(j) + \mu_i \cdot B_i(i) = \sum_{\{j=1\,|\,j\neq i\}}^{n} \mu_j \cdot \vec{b}_j + \mu_i \cdot \vec{v}$$

   requires multiple other lemmas to be accepted in Waterproof. In particular, the system requires a proof that $B_i(i) = \vec{v}$, a proof that $B_i(j) = \vec{b}_j$, and another proof to establish that $\sum_{\{j=1\,|\,j\neq i\}}^{n} \mu_j \cdot B_i(j) = \sum_{\{j=1\,|\,j\neq i\}}^{n} \mu_j \cdot \vec{b}_j$ before accepting the step.

2. While we only use one large chain of equalities in the handwritten version of the proof, we cannot do the same in Waterproof since every single step needs to be broken down into several assertions that need to be proven before the step is correctly verified. Consequently, every step takes a considerable amount of lines and the proof progresses slower.

3. In the handwritten version, we write that 'it follows from linear independence ...' to obtain a crucial result in our proof. In Waterproof however, this is too large of a step to take. Namely, in order to use linear independence of the the vectors $(\vec{b}_1, \ldots, \vec{b}_n)$, we must first prove to the system that we can express the sum as a linear combination of the vectors $(\vec{b}_1, \ldots, \vec{b}_n)$. This step required some clever tricks and rewriting steps to establish in Waterproof.

4. Even though we succeed in expressing the sum as a linear combination of the vectors $(\vec{b}_1, \ldots, \vec{b}_n)$, we are still not finished. Once again we must rewrite, simplify, and use helper functions to manipulate expressions to finally obtain that $\mu_j = 0$ for every index $j \in \{1, \ldots, n\}$.

We think that this code could be optimized and reduced in length in future work by increasing the level of automation for lower levels of abstraction. This would help the system automatically verify simpler steps such as the rewriting steps mentioned above.

Second Half of the Proof of Lemma 4.26

Now that we have gotten linear independence out of the way, we are left with showing that B_i forms a generating system of the vector space V. If we recall the definition from Section 4.3.1, it suffices to show that every vector in V can be written as a linear combination of vectors of the tuple B_i.

Handwritten Proof Version

Suppose that we were again using pen-and-paper to prove this statement. Our proof might be written as follows:

Proof: Using the fact that any vector in V can be written as a linear combination of the vectors $(\vec{b}_1, \ldots, \vec{b}_n)$, we know that it suffices to show that $\vec{b}_i$ is generated by B_i. Now since $v = \sum_{k=1}^{n} \lambda_k \cdot \vec{b}_k$, we can write $\vec{b}_i = \lambda_i^{-1} \cdot \left(\vec{v} - \sum_{\{k=1\,|\,k\neq i\}}^{n} \mu_k \cdot \vec{b}_k\right)$. Using that $\vec{v}$ is generated by B_i (and since of course for every index $j \neq i$, $\vec{b}_j$ is generated by B_i), it also holds that $\vec{b}_i$ is generated by B_i. Then we conclude that every vector in V is generated by B_i and hence B_i is a generator of V.

Proof in Waterproof

Similar to the proof for the first half, the proof of the second half in Waterproof is a little bit more involved than the handwritten equivalent. We include a brief summary of comparison between the two proofs below:

1. In the handwritten version, we write

$$\vec{b}_i = \lambda_i^{-1} \cdot \left(\vec{v} - \sum_{\{k=1 \,|\, k \neq i\}}^{n} \mu_k \cdot \vec{b}_k \right)$$

   and leave it to the readers to verify the correctness of the statement. In Waterproof however, it is the automation system and not the reader that verifies the correctness of step. Sometimes this system needs a little bit of assistance in order to accept such a step, which is why it is not so simple to jump to these intuitive conclusions.

2. Where in the handwritten version we simply justify that $\vec{b}_i$ must be generated by B_i based on some things we know to be true, Waterproof requires that we actually prove that this is true. In particular, this involves choosing an appropriate sequence of scalars $(\lambda_0, \lambda_1, \ldots, \lambda_{n-1}) \in X$ such that we can write $\vec{b}_i = \sum_{j=1}^{n} \lambda_j \cdot B_i(j)$. We do this by defining and creating several definitions and helper functions to create and compute an appropriate sequence of scalars. Taking these into account, proving this isolated statement ended up taking a little over 50 lines code.

3. While the handwritten proof only needs one line to conclude that every vector in the vector space can be generated by B_i, it should be no surprise that this statement did not meet the standard of mathematical rigor enforced by the automation engine. To reach this conclusion, we needed to be able to show that any vector $\vec{w} \in V$ could be written as a linear combination of vectors from B_i. In the most concise of terms, we did this as follows: we take an arbitrary vector $\vec{w}$ and express it as a linear combination of the vectors $(\vec{b}_1, \ldots, \vec{b}_n)$. Then we rewrite this linear combination by splitting it up into two terms that (we can prove) are generated by B_i. After proving that both terms are generated by B_i (both of these statements are essentially individual lemmas), the conclusion follows.

Again, for full access to the proof in Waterproof, see the file *Lemma_4_26.v*. Close inspection of this file highlights that whenever possible, the structure of the typed proofs resembles the style of handwritten proofs. Regrettably, there were some specific instances of Coq tactics that could not be translated into the declarative proof style that we tried to maintain throughout all of our work. In particular, the Lemmas *split_other_index* and *eq_refl* could not be adapted into readable notation, and hence we opted to stick with the procedural Coq tactics for these instances. For instance, we use the Lemma *split_other_index* to 'split' sums on the index i in the following manner:

$$\sum_{i=1}^{n} \vec{v}_i = \sum_{\{j=1 \,|\, j \neq i\}}^{n} \vec{v}_j + \vec{v}_i$$

We expected to be able to apply this tactic by typing the following line into Waterproof:

```
By split_other_index it holds that (...)
```

But for a reason we could not quite determine, the automation system did not accept this as a valid step. Instead, applying the (equivalent) Coq tactic

```
rewrite (split_other_index n _ i).
```

allows us to reformulate our goal into the format that we want as a substitution for the more declarative Waterproof tactic. While at times this compromised the flow of the proof style, 'sneaking' in these Coq tactics allows us write the proofs in a more efficient manner.

4.3.3 Lemma 4.28

With the conclusion of Lemma 4.26, we still had a small amount of time left to achieve more work in this report. Instead of devoting time on improving existing results, we decided to aim for an even more 'complicated' result. Namely, we attempted to try and prove Lemma 4.28 from the lecture notes [Heckenberger, 2023]. Given that this lemma is based on the result from Lemma 4.26, it was unsurprising that this lemma was more complicated to prove in Waterproof than anything we had proven thus far. In particular, this result (also known as the Steinitz exchange theorem) goes as follows:

Lemma 4.28. Let $k, n \in \mathbb{N}$, and let $\vec{a}_1, \ldots, \vec{a}_k \in V$, and $\vec{b}_1, \ldots, \vec{b}_n \in V$ be vectors in a vector space V. If $(\vec{a}_1, \ldots, \vec{a}_k)$ are linearly independent, and if $(\vec{b}_1, \ldots, \vec{b}_n)$ forms a basis of V, then

(1) $k \leq n$

(2) there exist vectors $\vec{c}_{k+1}, \ldots, \vec{c}_n \in \{\vec{b}_1, \ldots, \vec{b}_n\}$ such that the tuple $(\vec{a}_1, \ldots, \vec{a}_k, \vec{c}_{k+1}, \ldots, \vec{c}_n)$ also forms a **basis** of V.

Working on the formal verification of this induction proof in Coq was nontrivial. In fact, this piece of formal verification proved to be much more complex then we had anticipated. Requiring many additional lemmas and the use of advanced Coq techniques, we spent several weeks trying to merely establish the base case of induction (i.e. that by replacing the first zero elements of the basis $(\vec{b}_1, \ldots, \vec{b}_n)$ by a set of zero independent vectors, we still obtain a basis of V).

The detailed implementation of our proof proved to be complex and time-consuming, and was largely dependent on establishing the existence of a specific set of vectors $\vec{c}_{k+1}, \ldots, \vec{c}_n$ under certain conditions related to linear independence. There were multiple factors that made the formalization of this proof so much more difficult than others, which we briefly discuss below. We should also mention here that we attempted to prove a slightly simplified version of this proof. Namely, we worked under the assumption that we could use first condition $(k \leq n)$ without proving it.

Base Case

The very first challenge in this proof involved finding a way to join two tuples of vectors together in order to obtain a new tuple of vectors. For instance, explicitly asserting that the first k vectors of the tuple $(\vec{a}_1, \ldots, \vec{a}_k, \vec{c}_{k+1}, \ldots, \vec{c}_n)$ belong to the set of vectors $\vec{a}_1, \ldots, \vec{a}_k$ was one aspect that we needed to enforce.

The reason why this proved to be challenging is due to the unrelenting rigor of Coq's automation system in these higher levels of abstraction. To illustrate, suppose we have a tuple of n vectors, denoted by B.

$$B = (\vec{b}_1, \ldots, \vec{b}_n)$$

It is not difficult to see that if we were to replace the first **zero** elements of this tuple by a different tuple of zero vectors, then we would end up with the exact same tuple B, because we essentially did nothing. The system, however, identified that there was a distinction between the types of this tuple B and the split between this tuple B and the tuple of zero vectors. In other

words, it would view I_n and I_{n-0} as different objects, even though $(n = n - 0)$.

After setting up some structure to deal with type equivalences, such as a lemma proving that $(I_n = I_{n-0})$, and using the documentation provided by the *fintype* library, we figured out how to address these challenges. In particular, we used folded mappings to continuously change the system's type interpretation. For instance, in the code snippet below, the folded mapping

```
@cast_ord (0 +(n-0)) n (esym (eq_n_k_plus_n_min_k(n, 0%N, H)))
```

allows for interpreting an element from a tuple of (n) vectors as an element from a tuple of $(0 + (n - 0))$ vectors instead. We will not go into the full details, but notice that the code below should be read as follows: 'Splitting the tuple $B = (\vec{b}_1, \ldots, \vec{b}_n)$ with the tuple A consisting of zero vectors, is equivalent (index-wise) to the tuple $B = (\vec{b}_1, \ldots, \vec{b}_n)$.'

```
my_split_2(0%N, n, H, A, B \o (@cast_ord (0 +(n-0)) n
           (esym (eq_n_k_plus_n_min_k(n, 0%N, H)))), i) = B i.
```

Although it may not look pretty, using these folded mappings relying on type equivalence allows us to work with higher levels of abstraction.

After having dealt with the issues regarding type interpretations and type equivalence, we were almost finished with the base case of the induction proof. Setting up a few extra helper functions and lemmas (such as the Lemma *eq_bases* to assert that index-wise equality is a sufficient condition for comparing two bases), we finished our completed proof of this first step. For more details, see the file *MySplits.v*.

Induction Step

Given that the (seemingly trivial) base case of induction proved to be rather complicated, it should be no surprise that the induction step required even higher levels of abstraction and complexity. Unfortunately, we were unable to complete this step, partly because of the complexity of the abstraction, but mostly due to time constraints. The file *MySplits.v* outlines a plan for finishing this proof (using the result of Lemma 4.26), but does not offer a complete solution.

Future work will first need to face the challenge of removing/adding vectors to tuples of vectors in order to complete this proof. For instance, showing that the existence of (n) linearly independent vectors implies the existence of $(n-1)$ linearly independent vectors is one such result that we could not establish under time constraints. We faced significant difficulty in working with equivalent types of tuples of vectors (e.g. the equivalence of the tuple of (n) vectors and the same tuple of $(n+0)$ vectors), so we predict that working with different types of tuples of vectors might prove to be even more so.

Reflection

While it took a considerable amount of time and effort, we succeeded in proving Lemma 4.26 in its entirety. The proof is complete and we consider this to be the greatest achievement in this report. Completing this proof demonstrates that Waterproof as a proof verification engine can work with higher levels of abstraction and algebra.

However, when we attempted to reach even higher levels of abstraction with Lemma 4.28,

we were ultimately unable to completely finish the proof in Waterproof. We repeatedly ran into difficulties with respect to the strictness of the automation system, and could not figure out a way to address every challenge before the deadline of this report. Nonetheless, we are optimistic that given more time, we could find a way to work around these challenges and complete this proof as well.

We should make note here that there is still ample room for improvement with the automation system, the presentation of the code, accessibility, etc. In future work we would recommend the introduction of more intuitive and user-friendly language, replacing phrases and syntax such as

```
By base_definition it suffices to show that
  (linearly_independent(n, replace_index(n, B, i, v))
  AND generator_of_V(n, replace_index(n, B, i, v))).
```

by something that is easier for new users to read, such as

```
By the definition of a base it suffices to show that
  (B_i is linearly independent and B_i is a generator of V).
```

Nevertheless, we are very pleased to demonstrate that Waterproof is capable of working with these advanced algebraic concepts.

Chapter 5

Discussion

As we established the foundations of linear algebra topic by topic, and constructed a large structure of support through lemmas and Theorems, we found that Waterproof's automation system generally performed well. The proof structure and explicit steps behaved as expected, with Waterproof readily accepting logic based on our set of specified rules. This facilitated smooth progress in establishing notions of fields (Section 3.2.2) and vector spaces (Section 3.3.2).

After making some minor modifications to Waterproof's underlying system, such the introduction of classical logic and improved functionality of logical tactics, we believe Waterproof could readily be applied to help teach students write well-structured proofs for this introductory part of the course. Despite some differences between the notation of handwritten proofs and their Waterproof counterparts, such the '$\in$' symbol being replaced by ':', or the symbols for vector addition '$+$:' and scalar multiplication '$*$:' being unconventional, they are not so significant as to cause confusion or frustration for university-level students. In fact, we hope that the slightly more mechanical structure of the proofs in Waterproof might actively enhance user's understanding of algebraic proofs.

In Chapter 4 we demonstrated how the *mathcomp* libraries could be used to build up a variety of algebraic theory. By adapting these result into our notions of fields and vector spaces, we proved a series of lemmas that combined to construct the proofs for complex results such as Lemma 4.26. We proceeded to explore higher levels of abstraction by introducing the notions of splitting and exchanging tuples of vectors with Lemma 4.28. We did not quite prove the completed version of this latter lemma in Waterproof, but we successfully finished the first steps. Although we did not present all of these proofs in a format optimized for educational purposes, the fact that we were able to establish them in the first place demonstrates Waterproof's capacity to handle advanced theoretical aspects of Linear Algebra.

Currently, our evaluation of Waterproof's benefits for teaching Linear Algebra has been based on our own personal experience, and that of the report's supervisors; J.W. Portegies (developer of Waterproof) and I. Heckenberger (lecturer for Linear Algebra at the University of Marburg). While we have certainly noticed a significant improvement in our ability to write mathematical proofs, the time and effort invested by us in this report necessarily creates a bias towards our judgement of Waterproof's effectiveness. In a future study, it would be good to investigate Waterproof's teaching effectiveness in a systematic way. One possibility includes a control group of students using Waterproof with another group of students neglecting its use, with an evaluation based on student feedback, a performative comparison of grades, and an analysis of both groups' handwritten proof structure.

5.1 Future improvements

Though the updated version of Waterproof has greatly increased support for algebraic concepts, there are still many possible improvements to be made as mentioned earlier in Section 4.3.3. We will briefly discuss a few potential improvements here.

Automation system

Future versions of Waterproof could further optimize the flexibility of the automation system. At present, the system can appear to be too strict in some cases, refusing to accept some seemingly trivial steps. Consider for instance the following case: suppose that we have an element $x \in X$ in a field and we are working under the assumption that $x \neq 0$. Then we know for a fact that there exists a multiplicative inverse such that $x \cdot x^{-1} = 1$. If we were to enter the following (equivalent) statement in Waterproof however, the command fails because the system cannot find a proof that this is true.

```
It holds that (x^{-1} * x = 1).
```

Clearly $x^{-1} \cdot x = x \cdot x^{-1} = 1$, so we would not expect the system to have an issue with this statement. A way to get around this issue is to first assert that $x^{-1} \cdot x = x \cdot x^{-1} = 1$ by explicitly entering this command, after which the system will accept the statement $x^{-1} \cdot x = 1$.

In order to keep things efficient and avoid needing too many extra lines of code to prove something, the system continuously needs a higher level of abstraction as the complexity of the proofs increases, with increased automation in the lower levels of abstraction. For illustrative purposes, consider the following example.

Suppose that we have two vectors $\vec{v}$ and $\vec{w}$ in a vector space V, and suppose that they are both generated by a basis of V. In Section 4.3.1, we proved that any linear combination of the two vectors is then also generated by that same basis. Therefore, it holds also holds that $(\vec{v} + \vec{w})$ is generated by this basis. However, if we were to enter the line equivalent to

'It holds that $\vec{v} + \vec{w}$ is generated by this basis'

in Waterproof, then the command fails. The reason for this is as follows: the system is able to verify that $(1 \cdot \vec{v} + 1 \cdot \vec{w})$ is generated by the basis, but needs an additional step asserting the equivalence $(1 \cdot \vec{v} + 1 \cdot \vec{w} = \vec{v} + \vec{w})$ before it will accept the aforementioned statement. In these cases it would be preferable for users if the system could automatically make such basic steps.

We recognize that continuously building up the structure for these increasing levels of abstraction is a lot of work. However, the potential benefits could vastly improve the readability and efficiency of these higher-level proofs. There has been some work on formalizing complexity theory in proof assistants, but this area of research has not been explored extensively ([Wiedijk, 2023], [Gäher and Kunze, 2021]). A more detailed analysis of the complexity of the Coq automation system falls outside of the scope of this report, but could be an interesting avenue to explore in future work.

Accessibility

The notations that we used in our work to define more complicated algebraic notions, such as linear independence of vectors in Section 4.3.1, could be re-designed to vastly improve the readability of our proofs in Waterproof. For instance, if we wanted to establish that the vectors $(\vec{v}_1, \ldots, \vec{v}_n)$ are linearly independent, then we would type the following line in Waterproof:

```
It holds that (linearly_independent(n, F))
```

Here $F : I_n \mapsto V$ is a function that maps the index $i \in [0, \ldots, n-1]$ to the vector $\vec{v}_i \in [\vec{0}, \ldots, \vec{v}_{n-1}]$, which should be interpreted as the tuple of vectors $(\vec{v}_1, \ldots, \vec{v}_n)$. This notation could be re-designed in such a way that the structure more closely resembles handwritten conventions. For instance, the concepts might be easier to grasp and work with for new students if they could simply type:

```
It holds that (v_1, . . . , v_n) are linearly independent.
```

Another example is the prevalence of Coq tactics that have not yet been adapted into Waterproof's procedural style. For instance, if we want to introduce the tuple of vectors $(\vec{v}_1, \ldots, \vec{v}_n)$, then we type

```
Take F : ('I_n -> V).
```

Here F is again the function described above. This command would make more sense if students could instead type similar to the following:

```
Let (v_1, ..., v_n) : V.
```

Chapter 6

Conclusion

Since their inception around the 1970s, proof assistants—computer programs that help humans create formal proofs—have grown increasingly popular. Though initially not created for education purposes, the potential benefits of integrating these tools with traditional teaching of proof-writing is significant. As these automation systems become more user-friendly, they may simplify the process of learning mathematical proof-writing. For instance, the educational software Waterproof (an extension of the proof assistant Coq) has successfully supplemented teaching of the course Analysis 1 at the TU/e.

In this report we explored the potential future of Waterproof as a useful tool in teaching Linear Algebra, and laid the groundwork for future studies in this area. We were successful in proving a range of both simple and more complicated algebraic results, which is the first step towards using Waterproof in an educational setting. Our results and substantial evidence suggest that Waterproof can effectively handle introductory algebraic theory. However, the formalization of more complex algebraic theory for educational purposes requires further development to ensure that Waterproof is sufficiently user-friendly for high-level formal proofs. Regardless, with its emphasis on mathematical rigor and a powerful verification engine, Waterproof demonstrates great potential as a teaching tool for proof-writing in Linear Algebra.

Our report presents the use of Waterproof for educational support and automated proof-verification in mathematics, providing a solid foundation for future research. Our work, accessible via this Github repository, could assist in the creation of educational modules for first-year students taking Linear Algebra at the University of Marburg, helping them hone their proof-writing skills.

We hope that this the ideas discussed in this report evoke enthusiasm for Waterproof as an educational software and improve its accessibility for mathematicians. Even though some may argue against the ubiquity of automation, we hope that the development of automated theorem solvers as educational tools will become of increased interest to the mathematical community. As automation increasingly influences various aspects of life, we foresee it becoming an invaluable tool for discovering, proving, and teaching mathematics.

Bibliography

[Aldrich et al., 2008] Aldrich, J., Simmons, R. J., and Shin, K. (2008). Sasylf: An educational proof assistant for language theory. In *Proceedings of the 2008 International Workshop on Functional and Declarative Programming in Education*, FDPE '08, page 31–40, New York, NY, USA. Association for Computing Machinery. 3, 4, 6

[Barras et al., 1997] Barras, B., Boutin, S., Cornes, C., Courant, J., Filliâtre, J.-C., Giménez, E., Herbelin, H., Huet, G., Muñoz, C., Murthy, C., Parent, C., Paulin-Mohring, C., Saïbi, A., and Werner, B. (1997). The Coq Proof Assistant Reference Manual : Version 6.1. Research Report RT-0203, INRIA. Project COQ. 4, 9

[Bartzia et al., 2022] Bartzia, E., Meyer, A., and Narboux, J. (2022). Proof assistants for undergraduate mathematics and computer science education: elements of a priori analysis. In Trigueros, M., editor, *INDRUM 2022: Fourth conference of the International Network for Didactic Research in University Mathematics*, Hanovre, Germany. Reinhard Hochmuth. 3

[Bastiaansen, 2023] Bastiaansen, B. (2023). History of the proof assistant coq. 2WH60 research paper. Eindhoven University of Technology. 3

[Bertot, 2006] Bertot, Y. (2006). Coq in a hurry. *arXiv*. 4

[Bertot et al., 2008] Bertot, Y., Gonthier, G., Ould Biha, S., and Pasca, I. (2008). Canonical big operators. In Mohamed, O. A., Muñoz, C., and Tahar, S., editors, *Theorem Proving in Higher Order Logics*, pages 86–101, Berlin, Heidelberg. Springer. 23, 24, 25

[Bertot and Pierre, 2004] Bertot, Y. and Pierre, C. (2004). *Interactive theorem proving and program development: Coq'art: The calculus of Inductive Constructions*. Springer. 1

[Beurskens, 2019] Beurskens, T. P. J. (2019). Computer programs for analysis. Bachelor's thesis, Eindhoven University of Technology. Retrieved March 1, 2023, from `https://research.tue.nl/en/studentTheses/computer-programs-for-analysis`. 1

[Cohen et al., 1999] Cohen, A. M., Cuypers, H., and Sterk, H. (1999). *Algebra interactive! : learning algebra in an exciting way*. Springer, Berlin ;. 7

[Coquand and Huet, 1985] Coquand, T. and Huet, G. (1985). Constructions: A higher order proof system for mechanizing mathematics. In Buchberger, B., editor, *EUROCAL '85*, pages 151–184, Berlin, Heidelberg. Springer Berlin Heidelberg. 1

[Cruz-Filipe et al., 2004] Cruz-Filipe, L., Geuvers, H., and Wiedijk, F. (2004). C-corn, the constructive coq repository at nijmegen. In Asperti, A., Bancerek, G., and Trybulec, A., editors, *Mathematical Knowledge Management*, pages 88–103. Springer Berlin Heidelberg. 6

[de Bruijn, 1994] de Bruijn, N. (1994). Generalizing automath by means of a lambda-typed lambda calculus**reprinted from: Kueker, d. w., lopez-escobar, e. g. k. and smith, c. h., eds., mathematical logic and theoretical computer science, p. 71-92, by courtesy of marcel dekker inc., new york. In Nederpelt, R., Geuvers, J., and de Vrijer, R., editors, *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*, pages 313–337. Elsevier. 3

[de Moura et al., 2015] de Moura, L., Kong, S., Avigad, J., van Doorn, F., and von Raumer, J. (2015). The lean theorem prover (system description). In Felty, A. P. and Middeldorp, A., editors, *Automated Deduction - CADE-25*, pages 378–388, Cham. Springer International Publishing. 6

[Dénès et al., 2012] Dénès, M., Mörtberg, A., and Siles, V. (2012). A refinement-based approach to computational algebra in coq. In Beringer, L. and Felty, A., editors, *Interactive Theorem Proving*, pages 83–98, Berlin, Heidelberg. Springer Berlin Heidelberg. 1

[Gäher and Kunze, 2021] Gäher, L. and Kunze, F. (2021). Mechanising Complexity Theory: The Cook-Levin Theorem in Coq. In Cohen, L. and Kaliszyk, C., editors, *12th International Conference on Interactive Theorem Proving (ITP 2021)*, volume 193 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 20:1–20:18, Dagstuhl, Germany. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. 36

[Geuvers, 2009] Geuvers, H. (2009). Proof assistants: History, ideas and future. *Sadhana*, 34(1):3–25. 3, 6

[Gonthier, 2005] Gonthier, G. (2005). A computer-checked proof of the four colour theorem. 3

[Gonthier et al., 2007] Gonthier, G., Mahboubi, A., Rideau, L., Tassi, E., and Théry, L. (2007). A modular formalisation of finite group theory. In Schneider, K. and Brandt, J., editors, *Theorem Proving in Higher Order Logics*, pages 86–101, Berlin, Heidelberg. Springer Berlin Heidelberg. 23

[Gugerty, 2006] Gugerty, L. (2006). Newell and simon's logic theorist: Historical background and impact on cognitive modeling. *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, 50:880–884. 3

[Heckenberger, 2023] Heckenberger, I. (2023). Vorlesung lineare algebra 1. University Lecture Notes. Retrieved April 8, 2023, from `https://www.mathematik.uni-marburg.de/modulhandbuch/20181/BSc_Wirtschaftsmathematik/Grundlagen_der_Mathematik/Lineare_Algebra_I.html`. 1, 2, 8, 12, 14, 16, 17, 21, 32

[Huet, 1992] Huet, G. (1992). The gallina specification language: A case study. In Shyamasundar, R., editor, *Foundations of Software Technology and Theoretical Computer Science*, pages 229–240, Berlin, Heidelberg. Springer Berlin Heidelberg. 3

[Inria, 1995] Inria, C. (1995). Early history of coq. Retrieved May 23, from `https://coq.inria.fr/refman/history.html`. 1

[Knobelsdorf et al., 2017] Knobelsdorf, M., Frede, C., Böhne, S., and Kreitz, C. (2017). Theorem provers as a learning tool in theory of computation. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*, ICER '17, page 83–92, New York, NY, USA. Association for Computing Machinery. 4

[Mahboubi and Tassi, 2021] Mahboubi, A. and Tassi, E. (2021). *Mathematical Components*. Zenodo. 3, 7, 23

[Nair and Singh, 2018] Nair, M. T. and Singh, A. (2018). Linear algebra. 21

[Newell and Simon, 1956] Newell, A. and Simon, H. (1956). The logic theory machine–a complex information processing system. *IRE Transactions on Information Theory*, 2(3):61–79. 3

[Poernomo et al., 2007] Poernomo, I. H., Wirsing, M., and Crossley, J. N. (2007). *Adapting proofs-as-programs : the Curry - Howard protocol*. Springer New York. pages 5-6. 3

[TU/e, 2003] TU/e (2003). The automath archive. Retrieved May 20, from `https://www.win.tue.nl/automath/`. 3

[Wemmenhove et al., 2022] Wemmenhove, J., Beurskens, T., McCarren, S., Moraal, J., Tuin, D., and Portegies, J. (2022). Waterproof: educational software for learning how to write mathematical proofs. Retrieved March 1, 2023, from https://arxiv.org/abs/2211.13513. 1, 3, 4, 6

[Wiedijk, 2003] Wiedijk, F. (2003). Comparing mathematical provers. In Asperti, A., Buchberger, B., and Davenport, J. H., editors, *Mathematical Knowledge Management*, pages 188–202, Berlin, Heidelberg. Springer Berlin Heidelberg. 1

[Wiedijk, 2023] Wiedijk, F. (2023). Formalizing 100 theorems. Retrieved May 24, from http://www.cs.ru.nl/~freek/100/. 3, 36

Appendix A

Executive Summary

This report is a detailed analysis on the use of Waterproof, a custom Coq library and editor designed for educational purposes, in the mathematical discipline of Linear Algebra. In short, the report is organized into several chapters:

1. Introduction: Discusses the concept of linear algebra, proof assistants, and the Coq proof assistant. It introduces Waterproof, a tool designed to make Coq proofs more similar to pen-and-paper proofs, and provides an overview of the report.
2. Related Work: Compares alternative methods and proof assistants implemented in an educational setting.
3. Fields and Vector Spaces in Waterproof: Covers the introductory concepts of fields and vector spaces, and their initialization in Waterproof. It provides various examples of proofs and exercises that can be applied as learning aids for the course Linear Algebra.
    - Though there are some slight modifications to the automation system that might be desirable, this chapter demonstrates that Waterproof can handle any proof covering the introductory material of the course Linear Algebra (or at least any proofs attempted in this project).
4. Expanding on Algebraic Theory in Waterproof: Discusses the construction of support for algebraic concepts that Waterproof was not yet equipped for, such as linear independence, bases of vector spaces, etc.
    - After working extensively on developing the formalization of these algebraic concepts, Waterproof is able to handle many complex algebraic statements.
    - It should be noted that the work required to establish any of these statements in Waterproof takes significantly more effort when compared to writing down an equivalent proof on paper.
    - Despite expressing confidence that, given enough time and support, Waterproof can handle any of the theory covered in this report, some lemmas did not obtain a (complete) formal proof. In particular, the induction step of Lemma 4.28 was left with a partly incomplete proof.
    - Being able to 'handle' proofs of complex algebraic concepts does not directly translate to Waterproof being useful in a class-room setting. Further development of user-friendly infrastructure is needed to convert this formalized theory into material that could be used as learning aids for students.
5. Discussion: Reviews the progress made in the project and suggests future improvements.
6. Conclusion: Summarizes the findings of the report.

The report showcases the potential of Waterproof as an educational proof assistant, particularly for students taking Linear Algebra. It provides evidence via proofs and hopes to build excitement for the use-case of Waterproof in a wider range of mathematical subjects, assisting new students with the writing of formal mathematical proofs.