Formalising Mathematics in Lean

Gaussian Elimination

by

Maria Joseph

A dissertation submitted in partial fulfillment of the requirements for the degree of

B.Sc Honours in Mathematics

2021-2024

Supervised by: Dr. Divakaran D
Evaluated by: Dr.Divakaran D., Dr.Mohan R. and Dr. Ajaykumar K

Formalising Mathematics in Lean

Gaussian Elimination

Maria Joseph

Abstract

Lean is a programming language and proof assistant that was developed by Microsoft Research. Over the past few years, it has gained popularity as a powerful tool for formal verification of mathematical ideas. The Mathlib library in Lean is an extensive mathematics library that was developed as a community-driven effort. The aim was to develop a library that includes a wide range of mathematical concepts, providing a platform for rigorous and verified mathematics.

Through this project, we worked to understand the different components included within Lean's foundational framework that is utilized while doing formal verification in Lean. Additionally, to also, contribute to the community by formalising a small part of undergraduate mathematics that is yet to be done in the Mathlib Library, specifically Gaussian Elimination. This has been done utilising the already existing framework of proofs and methods in Lean's library.

Dedication

To Amma, Appa, and George

Declaration

I certify that this work contains no material that has been accepted for the award of any other degree or diploma in my name, in any university or other tertiary institution and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text. In addition, I certify that no part of this work will, in the future, be used in a submission in my name, for any other degree or diploma in any university or other tertiary institution without the prior approval of Azim Premji University.

Signature: Maria Joseph
Date: 27 April 2024

Acknowledgments

I would like to express my deepest gratitude to my supervisor, Dr. Divakaran D., for his unwavering support, invaluable guidance, and endless patience throughout the entire process of researching and writing this thesis. His expertise, insightful feedback, and encouragement have been instrumental in shaping the direction and quality of this work.

I am also grateful to the members of my thesis evaluation committee, Dr. Mohan R. and Dr. Ajaykumar K., for their valuable feedback, constructive criticism, and scholarly insights that have enriched this thesis.

My sincere gratitude to the faculty members of the Mathematics Department of Azim Premji University for their mentorship, encouragement, and intellectual contributions. Their dedication to fostering academic excellence has been a constant source of inspiration.

I would like to extend my thanks to the School of Arts and Sciences at Azim Premji University for providing me with the opportunity to pursue the Honours Track. This has been pivotal in shaping my academic pursuits.

Special thanks are due to my colleagues and friends at Azim Premji University for their moral support and stimulating discussions that have helped me maintain perspective and motivation throughout this journey.

Finally, I want to extend my deepest appreciation to my family for their unwavering love, unwavering support, and understanding throughout this journey.

Contents

Chapter 1

Introduction

What is Lean?

Lean is a versatile and powerful programming language and interactive proof assistant. It was developed by Microsoft Research under the leadership of Leanardo de Moura in 2013. The key mottos were to provide an extensible, efficient, scalable, and expressive system. Lean is designed to provide a platform for formal verification of mathematics, to verify and develop computational and software systems, and lastly as a domain-specific language. Lean has gained popularity in both academia and industry for its ability to express complex mathematical concepts, verify the properties of programs, and formally prove the correctness of mathematical theorems. It is an open-source project that has been hosted on git-hub. The latest released version of the programming language is Lean 4.

Formal Verification

Formal verification or formalisation is the process of using logical and computational methods to establish and prove claims that are expressed in formal mathematical terms is known as formal verification. This can be applied not only to mathematical theorems but also to the underlying algorithms of hardware and software systems, and network protocols, to verify the correctness of a system.

In order to support a mathematical claim it is required to provide proof. The developments in logic have shown that most proof methods can be reduced to a small set of axioms or rules. Therefore, this brings us to how a computer or system can help with establishing a claim. It can either be by helping develop a proof or by helping verify the proof is correct. This can be seen in the form of automated and interactive theorem provers.

Automated Theorem Provers

The motive of automated theorem proving is to automatically generate proofs of a conjecture or mathematical theorem given a set of base known facts. It focuses on the "finding" aspect. For example, this includes something like a computer algebra system (CAS) or a symbolic algebra system (SAS). This is any mathematical software with the ability to manipulate mathematical expressions in a way similar to the traditional manual computations of mathematicians and scientists. For example, GeoGebra CAS is a computer algebra system. The main objective of automated theorem provers is speed and efficiency. While that is important it sometimes comes at the expense of soundness. It proves difficult to verify the correctness of the results.

Interactive Theorem Provers

Interactive theorem proving focuses on the "verification" aspect so that every claim is supported by a prior definition or theorem that can be traced back all the way down to set rules and axioms. The history of interactive theorem proving follows the development of systems so that the user is allowed to interact with the system at gradually higher levels of abstraction, getting further away from the axioms and closer to informal mathematics. Coq and Isabel are examples of interactive theorem provers.

Mathlib - LEAN's Mathematics Library

Lean aims to bridge this gap between interactive and automated theorem proving, by providing an automated framework of methods and tactics that involves user interaction.

The Mathlib library in Lean is a unified library of formalised mathematics that was built as a community-driven effort. formalising mathematics in Lean can be seen as computer programming. There is a small set of mathematical axioms that built the foundational framework. These have been developed and built upon to create an automated set of tactics and methods that assist in providing proof through user interaction.

We provide mathematical definitions, theorems, axioms, and proofs in a language that Lean can interpret and understand. In turn, Lean provides feedback and information, interprets the expressions guarantees the well-formedness of an argument, and ultimately determines and certifies whether a proof is correct or wrong. In Lean, every claim must be supported by a prior definition of a theorem and therefore given its small foundational framework, it is possible to verify the correctness and accuracy of proof at every step.

Lean has a relatively small trusted kernel, and the rich API allows users to export their developments to other systems and implement their own reference checkers.

Xena Project

A specific intersection of interests between undergraduate mathematics and LEAN came with the introduction of the Xena Project by Kevin Buzzard at Imperial College, London. The aim of this project was to introduce undergraduate students to computer verification software. Through this, a community driven effort has begun to digitise the various complex mathematical objects that are taught in undergraduate mathematics into Lean's mathematics library, Mathlib. It has also provided a platform for mathematicians, students, and enthusiasts to collaborate on formalising mathematical proofs and concepts in Lean.

A new interest in Lean has sparked in the mathematics community as an aftermath of the

Liquid Tensor Experiment by Peter Scholze. Just a year and a half after the problem was posed by Scholze, the Mathlib community was able to formalise and verify the entire main theorem in Lean Proof Assistant. The short time and efficiency with which this could be achieved was a breakthrough in the perception of proof assistants and theorem provers.

Chapter 2

Theorem proving in Lean

Type Theory

Type theory is a formal system for categorizing and working with objects based on their types. This provides a powerful and expressive way to express complex mathematical assertions, write complex hardware and software specifications, and reason about both of these naturally and uniformly. Lean works on the underlying notion of type theory.

In Lean everything has a type. Types for natural numbers, integers, sets, functions, etc. Types help ensure that objects are used consistently and that operations are performed on objects of appropriate types. As simple code in Lean can be expressed as seen below. The `def` command introduces a new constant into the working environment and the `#check` command gives you the type of any object.

```
def a1 : ℕ := 8 -- a is a natural number
def a2 : ℕ := 6
def b1 : Bool := True -- b is a boolean
def b2 : Bool := True

#check a1 -- output := Nat
#check b2 -- Bool
```

```
#check a1 + a2 -- Nat
#check b1 && b2 -- "&&" is the Boolean and
```

The power of type theory lies in the possibility of creating new types from existing ones. For example, if `a` and `b` are types, and `a → b` represents functions from `a` to `b`.

`a × b` is a new type and represents the cartesian product of `a` and `b`. This notion of type theory is also extended. ℕ and `Bool` are also objects and they too have a type. They are defined to be of type `Type`.

```
#check ℕ -- Type
#check Bool -- Type
#check ℕ → ℕ -- Type
#check Bool × ℕ -- Type
```

Lean also lets you define variables of various types rather than defining constants. These do not have a specific value associated unlike the constants defined using `def`. They are not computationally significant and are typically used in theorem statements or function definitions to reference a variable universally or existentially.

```
variable (a b : ℕ)
variable (c : Type)
variable (d : Bool)
variable (e : ℕ → Bool)

#check a + b -- ℕ
#check e -- ℕ → Bool
#check e b
-- you are applying a function from ℕ → Bool on an object of type ℕ so you will get
    Bool
#check d -- Bool
```

Functions as Lambda Abstraction

Lean provides a specific keyword `fun` or λ to define functions.

```
#check fun (x : ℕ) => x + 5    -- ℕ → ℕ
#check λ (x : ℕ) => x + 5    -- ℕ → ℕ (over here fun and λ mean the same thing)
```

#eval evaluates any given expression at a given parameter. You can evaluate a function at a given value as shown below.

```
#eval (fun (x : ℕ) => x + 5) 10 -- 15
```

Creating a function from an expression is known as lambda abstraction. It is a way of creating an anonymous function by specifying the parameters and its behavior. In our example above, `(x : ℕ)` is the parameter of the function and the body of the function is `x + 5`. The function is taking each value x of type natural numbers to x + 5.

In general terms,suppose you have a variable `x : α` and an expression `t : β`, the expression `fun (x : α) => (t:β)` can be seen as a function from $\alpha \to \beta$ that maps each value of `x` to a value of `t : β`.

Proposition as types

A proposition is a statement that is either true or false. They can be seen as mathematical statements. In Lean, propositions are also a type. The type associated with propositions is `Prop`. Let `p : Prop`, then the truth value of `p` can be seen the same as the type `p` being inhabited. That is, if there is any `hp : p`, then `p` is true. Therefore, providing a proof of a proposition would be equivalent to constructing an object of the same type. The relation between formal logic and type theory enables type-checking as a mechanism to verify the correctness of a logical proof. These ideas can be will be explained further as we go into theorem proving.

Various constructors have also been defined in Lean to be able to create new propositions from others.

```
variable (p q : Prop)
```

```
#check And      -- Prop → Prop → Prop
#check Or       -- Prop → Prop → Prop (it can be written as ∨)
#check Not      -- Prop → Prop (it can be written as ¬)
#check Implies --Prop → Prop (written as → )

#check p ∧ q      -- Prop
#check p ∨ q      -- Prop
#check ¬p         -- Prop
```

There is another interesting observation to be made. When using the implies constructor, having an implication from `p` to `q` (where `p` and `q` are propositions) can be seen as having a function that takes elements of type `p` to elements of type `q`. This is why the function constructor → is used in place of the `Implies` constructor.

Theorems and Proofs

In Lean, a theorem can be written using the `theorem` command. Once stating the theorem, the proof should be provided after the `:=`.

```
theorem t1 : p → q → p :=
fun (hp : p) => fun (hq : q) => hp
```

Intuitively the proof works by assuming `p` is true and `q` is true to trivially prove that `p` is true. We are creating an object of the same type as the theorem statement.

When broken down this can be seen as ∀ `p`, we can constuct an element of the type `q` → `p`. That is, we can create a function that takes objects of type `q` to `p`. If the type `p` is not inhabited then we have the empty function. If `p` is inhabited, then we take any arbitrary element hp : p. We send this to a function of type `q` → `p`.

To create the function from `q` to `p`, with respect to a particular `hp` we define the constant

function that takes any object in `q` to `hp`.

Lean acts as a type checker and confirms that the object we have created in proof `fun (hp : p) => fun (hq : q) => hp` is of type `p → q → p` and therefore has the same type as our hypothesis. Then the proof is seen as complete.

The library in Lean also has theorems that have already been proved that can be called to prove a new theorem.

```
theorem t2 : p → (q ∨ p) :=
fun (hp : p) => Or.inr hp
```

`Or.inr` is a right injection theorem that says that p is true `hp : p` implies that `q ∨ p` is true. This can be interpreted in type theory as given an object of type `p` you can create an object of type `q ∨ p`.

If you do not want to declare a new theorem in the working environment and still wish to see if a proof works, you can use the `example` command.

```
example : ∃ x : Nat, x > 0 :=
have h : 1 > 0 := Nat.zero_lt_succ 0
Exists.intro 1 h
```

The `have` command establishes an intermediary hypothesis that needs to be proved. In the above example, it is proved using the theorem `Nat.zero_lt_succ` that says zero is less that its successor. `Exists.intro` is a theorem takes a proof that `x` is true and `p x` is true (p evaluated at x) to provide a proof (construct an object of the type) that `∃ x, p x` is true.

Chapter 3

Inductive Types and Structures

Inductive Types

As we have observed Lean's main library contains various types. These may be foundational types such as `Prop`, `Type`, `Type 1` etc. There are also additional types that have been defined such as `Nat`, `Bool`, `List` etc. In Lean, most concrete types and every constructor type other than dependant arrows is an instance of a general family of type constructions known as inductive types.

Inductive definitions allow you to create data structures by specifying their constructors and properties. Within these definitions, you have constructors that define the valid ways to create objects of the inductive type.

For example, let us look at the inductive type definition of natural numbers.

```
inductive Nat where
  | zero : Nat
  | succ : Nat → Nat
```

Over here we name the type of natural numbers to be `Nat`. It is constructed with the help of two constructors. Namely `zero` and `succ`.

Once defining the inductive type you can define axioms, properties, and theorems that characterise these objects.To better understand how the inductive type captures the entire natural numbers set, let us look at the addition function.

```
def add (m n : Nat) : Nat :=
  match n with
  | Nat.zero   => m
  | Nat.succ n => Nat.succ (add m n)
```

We define addition by fixing `m` and doing recursion on `n`. In the base case,`n` takes the value `Nat. zero` and we define `add m zero` as `m`. In the successor step, `n` takes the value `Nat.succ n` and we define `add m (succ n)` to be `succ (add m n)`.

From the definition of the inductive type and how it has been utilised to define addition, it is simple to infer what the `succ` constructor is doing. It takes any element `x : Nat` and spits out `x + 1 : Nat`. It is clear to see that once you have zero and such a function defined, we can generate the entire natural numbers type. Similarly, properties such as subtraction, multiplication, and all can be defined on natural numbers. Once you have axioms and properties defined, you can go on to prove theorems that follow from these definitions.

Another inductive type we have seen is `Or` while looking at propositions. The definition for this can be done as follows,

```
inductive Or (a b : Prop) : Prop where
  | inl : a → Or a b
  | inr : b → Or a b
```

Over here, we can see that an or statement or an expression of the `Or` can be constructed using two constructors. A proof that `a` is true can give you `Or a b` using the `Or.inl` constructor. Similarly, a `b` is true can give you `Or a b` using the `Or.inr` constructor.

`And` can also be defined. This can be done using just a single constructor.

```
inductive And (a b : Prop) : Prop where
  | intro : a → b → And a b
```

Given a proof that `a` is true and a proof that `b` is true, you can construct an object of the type `And a b` using the `And.intro` constructor.

In our examples previously we have also seen the use of the $\times$ operator that signifies cartesian product. The inductive type for it is called `Prod`

```
inductive Prod (α : Type u) (β : Type v) where
  | mk (fst : α) (snd : β) : Prod α β
```

It has one constructor that takes in two argument `(fst : α)` and `(snd : β)` and creates `Prod α β` or $\alpha \times \beta$

Structures

Lean's foundational system includes inductive types. These definitions may be recursive, as we have seen in the case of natural numbers. Or, they may have more than one constructor as seen for the `Or` inductive type. Inductive type that has only a single constructor and is non-recursive can be defined as structures. While formalising mathematics or writing programs, we often are required to define structures that contain many fields. The `structure` command in Lean allows you to do just that. It also allows you to define new structures based on previously defined ones.

Since the definition of `Prod` only utilised a single constructor and was non-recursive, this can also be defined as a structure.

```
structure Prod (α : Type u) (β : Type v) where
  mk :: (fst : α) (snd : β)
```

This example simultaneously introduces the inductive type, `Prod`, its constructor, `mk`, as well as the projections, `fst` and `snd`, as defined above.

Let us look at another example,

Transvections are linear transformations that add a multiple of one row to another row in a matrix. A transvection matrix is of such a form $A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & c \\ 0 & 0 & 1 \end{bmatrix}$ where c can at any place A_{ij} where $i \neq j$

Let T be a transvection matrix with c in the i-th row, j-th column. If you were to multiply a matrix M by T on the left, then c times the j-th row will bw added to the i-th row. For example, $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{bmatrix}$ x $\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$ = $\begin{bmatrix} 1 & 1 & 1 \\ 4 & 4 & 4 \\ 1 & 1 & 1 \end{bmatrix}$

All the information that is required to create a transvection matrix can be stored within a structure.

```
structure TransvectionStruct where
  (i j : n)
  hij : i ≠ j
  c : R
```

Over here, c is of type `R` and is in `i` row, `j` column of the matrix. We also have a proof that $i \neq j$. We will see later on that this structure will be easier to manipulate than directly working on a matrix since it has access to all the relevant fields. Converting the structure back into a matrix can also be done by defining a function.

Chapter 4

Tactic Mode

In Lean, a proof term is a representation of a mathematical proof. The tactic mode provides tactics and methods that can be seen as instructions or commands that explain how to construct a proof. In this mode, you deconstruct proofs into smaller goals and work on a single goal at each step. The major advantage of this mode is that at each stage you can see a visual representation of your proof state in the Lean info-view. Therefore, this takes away the hassle of having to keep track of different cases and parts of the proof in your memory.

Tactics play a very important role in theorem proving lean, listed below are some of the important tactics that we will be using moving forward.

- exact

  `exact e` closes the main goal if the type of the target is the same as `e`

  ```
  example (hp : p) : p := by
  exact hp
  ```

- apply

  `apply e` attempts to match the evaluated type of `e` against the current goal. If it succeeds then the goal is simplified

```
example (h: p → q) (hp : p) : q:= by
apply h
exact hp
```

By applying the hypothesis `p → q` on the goal to prove that `q` is true, the goal simplifies to it is sufficient to prove that `p` is true.

- intro

  The `intro e` tactic introduces a hypothesis named e, when possible.

```
example (hq : q) : p → q := by
intro hp
exact hq
```

- cases

  Let `x` be a variable or expression in the local context with an inductive type. `cases x` splits the main goal into a goal with respective to each constructor of the inductive type.

```
example (h : p ∨ q) : q ∨ p := by
cases h with
| inr hq =>  apply Or.inl hq
| inl hp =>  apply Or.inr hp
```

- rcases

  This tactic applies cases recursively. It pattern matches an inductive type with multiple arguments into different cases. example (p q r : Prop) (h : p q p): p q:= by rcases h with h1 —h2 — h3 exact Or.inl h1 exact Or.inr h2 exact Or.inl h3

- exists

  `exists x` is useful for existential goals. Over here `x` will be a specific witness for the existential quantifier that can be specified explicitly.

```
example (p q : ℕ  → Prop) : (∃ x, p x) → ∃ x, p x ∨ q x := by
  intro h
  cases h with
  | intro x px =>
  exists x
  apply Or.inl px
```

- rw

  `rw e` tactic stands for rewrite. It applies identity e as a rewrite rule to the target of the main goal. If e is preceded by the left arrow ($\leftarrow$ or $<-$), the rewrite is applied in the reverse direction. It also closes goals under reflexivity.

```
variable( a b c : ℕ )
example: (a = b) → (b = c) → (a = c) := by
intro ab bc
rw[ab,← bc]
```

- ext

  The `ext` tactic introduces anonymous variables wherever possible. The `ring` tactic over here is simplifying using the identities of a commutative ring

```
example : (fun x y : ℝ ↦  (x + y) ^ 2) = fun x y : ℝ  ↦ x ^ 2 + 2 * x * y + y
    ^ 2 := by
ext a b
ring
```

  In this example, we would like to show that equality holds for the functions evaluated at any value. Using `ext` we take arbitrary values a and b and show that equality holds for them.

- simp

  A group of identities in Lean's library have been listed with the attribute `[simp]`. The `simp` tactic uses these identities to attempt to rewrite the goal iteratively and simplify it.

```
example (x y z : ℕ ) : (x + 0) * (0 + y * 1 + z * 0) = x * y := by
simp
```

- push_neg

  This tactic pushes a negation if present into the conclusion of a hypothesis.

```
example  (p q : ℕ  → Prop) : ¬(∃ x, p x) → (∀ x ,¬ p x) := by
intro H
push_neg at H
exact H
```

- suffices

  This tactic allows you to restate the main goal with another expression of the same type. The expression is first considered as a hypothesis named `this` and you are required to prove the type consistency by showing that it is of the same type as the main goal. Once this is done, the main goal will change to the new expression.

```
example  : ¬(∃ (x: ℕ), 0 > x) := by
suffices ∀ x : ℕ , 0 ≤ x by push_neg;exact this
intro x
simp
```

- induction

  Assuming x is a variable in the local context with an inductive type, induction x applies induction on x to the main goal, producing one goal for each constructor of the inductive type.

```
example (x : ℕ) : x ≥  0 := by
induction' x with r IH
simp
```

```
apply (Nat.lt_succ.mpr) at IH
simp
```

- by_cases

  `by_cases p` makes a case distinction on p, resulting in two subgoals `h : p ⊢` and `h : ¬ p ⊢`

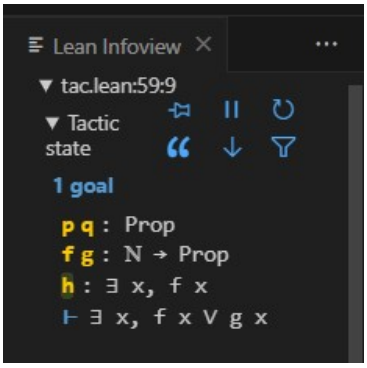
  ```
  example (h1 : ¬¬p) : p :=by
  by_cases p
  ·exact h
  ·apply absurd h h1
  ```

  Over here the absurd command gives that anything follows from two contradictory hypotheses.

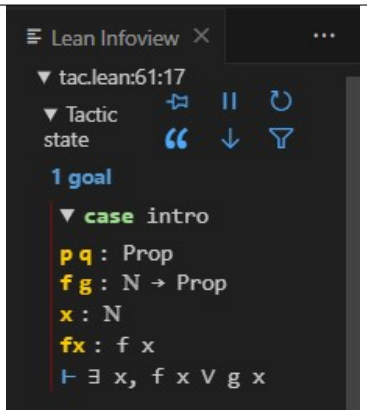
Let us look at this example once more, and break it down to understand how exactly the tactic mode works and assists in theorem proving.

```
example (p q : ℕ  → Prop) : (∃ x, p x) → ∃ x, p x ∨ q x := by
  intro h
  cases h with
  | intro x px =>
  exists x
  apply Or.inl px
```

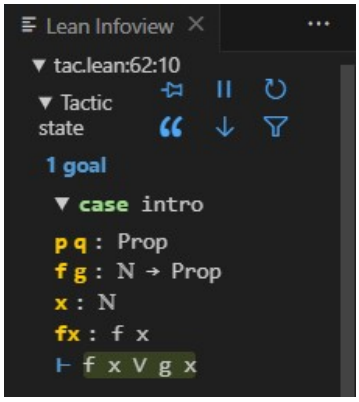
Before we begin the proof, on using the by command, Lean creates a tactic mode and states the goal to be proved



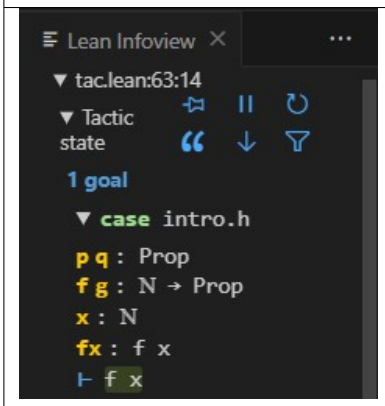
In line 1, using the intro command we assume the first part of the implication as hypothesis h.



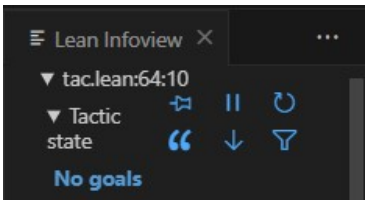
In line 2, we break down h into the two constructors of the ∃ type. This provides an instance of x where f x is true.



The exists command simplifies the goal by entering the x from h as the x required for the goal.



We have an Or statement as the goal and we know that we have the left constructor as one of the hypothesis. So we apply Or.inl



It can clearly be seen that the final goal is exactly the hypothesis fx, therefore closing the goal.

Chapter 5

Gaussian Elimination

In linear algebra, Gaussian elimination is an algorithm that is utilised for solving a system of linear equations. It consists of row-wise operations that are used on the corresponding matrix of coefficients to reduce the matrix into a triangular form.

A system of m equations in n unknowns is

$a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n = b_1$

$a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n = b_2$

...

...

...

$a_{m1}x_1 + a_{m2}x_2 + ... + a_{mn}x_n = b_m$

Over here the x_i's are the unknowns. And the a_i's and b_i's come from a field F.

The idea is to express the same system in the form $Ax = b$ where $A \in M_{m\times n}(F)$, $b \in F^m$ and $x \in F^n$. Particularly,

$$A = \begin{bmatrix} a_{11} & a_{12} & . & . & . & a_{1n} \\ a_{21} & a_{22} & . & . & . & a_{2n} \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ a_{m1} & a_{m2} & . & . & . & a_{mn} \end{bmatrix} \quad x = \begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ . \\ x_n \end{bmatrix} \quad b = \begin{bmatrix} b_1 \\ b_2 \\ . \\ . \\ . \\ . \\ b_m \end{bmatrix}$$

So given a matrix A and a vector b, we would like to solve for an unknown vector x.

Definition An $m \times n$ matrix A is said to be in row-echelon form if the nonzero entries are restricted to an inverted staircase shape. That is, it is an upper triangular or lower triangular matrix. In particular, we require,

- For an upper triangular matrix, $A_{ij} = 0$ when $i > j$
- For a lower triangular matrix, $A_{ij} = 0$ when $i < j$

The first step of the Gaussian Elimination process is to reduce the matrix A in the system $Ax = b$ into row-echelon form.

We would like to transform a system (A, b) to a system (C, d) with certain properties.

- A and C are matrices of the same size $m \times n$, over the same field F, and $b, d \in F^m$.
- The two systems $Ax = b$ and $Cx = d$ have the same solutions, that is, for any $x \in F^m$ we have $Ax = b$ is true if and only if $Cx = d$ is true.
- The matrix C is in row echelon form.

This process is carried out with the help of elementary row operations.

Definition Suppose $Ax = b$ is a system of m equations in n unknowns. An elementary row operation is one of the two procedures below.

1. Exchange the i-th equation and j-th equation.

   $(a_{j1}, a_{j2}, ..., a_{jn}) \rightsquigarrow (a_{i1}, a_{i2}, ..., a_{in})$, $b_j \rightsquigarrow b_i$

   $(a_{i1}, a_{i2}, ..., a_{in}) \rightsquigarrow (a_{j1}, a_{j2}, ..., a_{jn})$, $b_i \rightsquigarrow b_j$

   This can be represented by $E(i, j)$ where $1 \leq i, j \leq m$

2. Add a multiple c of i-th equation to the j-th equation.

   $(a_{j1}, a_{j2}, ..., a_{jn}) \rightsquigarrow (a_{j1} + ca_{i1}, a_{j2} + ca_{i2}, ..., a_{jn} + ca_{in})$, $b_j \rightsquigarrow b_j + cb_i$.

   This can be represented by $T(i, j, c)$ where $1 \leq i, j \leq m$

To each row operation, you can associate a $m \times m$ elementary row matrix.

$$E(i,j) = \begin{bmatrix} 1 & 0 & 0 & . & . & 0 \\ 0 & 1 & 0 & . & . & 0 \\ . & . & . & . & . & 0 \\ 0 & 1 & 0 & . & .. & 0 \\ 0 & 0 & 0 & . & 1 & 0 \\ 0 & 0. & & . & . & 1 \end{bmatrix}$$

Over here $E(i, j)$ is a matrix E such that

- $E_{kk} = 1$, $k \neq i, j$
- $E_{ij} = 1$ and $E_{ji} = 1$
- All other entries of the matrix are zeroes

$T(i, j, c)$ is the identity matrix with c appearing in the (i, j) position.

$$T(i,j,c) = \begin{bmatrix} 1 & 0 & 0 & . & . & 0 \\ 0 & 1 & 0 & c & . & 0 \\ . & . & . & . & . & 0 \\ 0 & 0 & 0 & 1 & .. & 0 \\ 0 & 0 & 0 & . & 1 & 0 \\ 0 & 0. & & . & . & 1 \end{bmatrix}$$

Proposition- Performing row operations on a system of linear equations is the same as multiplying by A and b on the left with the matrix corresponding to the row operation, in a system $Ax = b$.

Proof. For the first elementary operation, the matrix is given by

$$E(i,j) = \begin{bmatrix} 1 & 0 & 0 & . & . & 0 \\ 0 & 1 & 0 & . & . & 0 \\ . & . & . & . & . & 0 \\ 0 & 1 & 0 & . & .. & 0 \\ 0 & 0 & 0 & . & 1 & 0 \\ 0 & 0. & & . & . & 1 \end{bmatrix}$$

If you consider each row in the matrix as a vector in F^m, the collection of these vectors would give you a basis. In particular, if the standard basis for F^m is $[e_1, e_2..., e_i..., e_j, ..., e_m]$ then the matrix $E(i,j)$ can be written as

$$E = \begin{bmatrix} | & | & . & | & . & | & . & | \\ e_1 & e_2 & . & e_j & . & e_i & . & e_n \\ | & | & . & | & . & | & . & | \end{bmatrix}$$

This is the identity matrix $I_{m\times m}$ with the i-th and j-th rows switched.

$$A = \begin{bmatrix} a_{11} & a_{12} & . & . & . & a_{1n} \\ a_{21} & a_{22} & . & . & . & a_{2n} \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ a_{m1} & a_{m2} & . & . & . & a_{mn} \end{bmatrix} = \begin{bmatrix} | & | & . & . & | \\ A_1 & A_2 & . & . & A_n \\ | & | & . & . & | \end{bmatrix}$$

Here we are considering each column to be a vector in F^m. Then $E \times A$ is

$$\begin{bmatrix}\begin{bmatrix} | & | & . & | & . & | & . & | \\ e_1 & e_2 & . & e_j & . & e_i & . & e_m \\ | & | & . & | & . & | & . & | \end{bmatrix}\end{bmatrix} \times \begin{bmatrix} | & | & . & . & | \\ A_1 & A_2 & . & . & A_n \\ | & | & . & . & | \end{bmatrix} = \begin{bmatrix} | & | & . & . & | \\ E \times A_1 & E \times A_2 & . & . & E \times A_n \\ | & | & . & . & | \end{bmatrix}$$

For a particular k we have that,

$$E \times A_k = \begin{bmatrix} | & | & . & | & . & | & . & | \\ e_1 & e_2 & . & e_j & . & e_i & . & e_m \\ | & | & . & | & . & | & . & | \end{bmatrix} \times \begin{bmatrix} a_{1k} \\ a_{2k} \\ . \\ a_{ik} \\ . \\ a_{jk} \\ . \\ a_{mk} \end{bmatrix} = a_{1k}e_1 + a_{2k}e_2 + ... + a_{ik}e_j + ... + a_{jk}e_i + ... + a_{mk}e_m = \begin{bmatrix} a_{1k} \\ a_{2k} \\ . \\ a_{jk} \\ . \\ a_{ik} \\ . \\ a_{mk} \end{bmatrix}$$

Therefore we get that ,

$$E(i,j) \times A = E \times \begin{bmatrix} a_{11} & a_{12} & . & . & . & a_{1n} \\ a_{21} & a_{22} & . & . & . & a_{2n} \\ . & . & . & . & . & . \\ a_{i1} & a_{i2} & . & . & . & a_{in} \\ . & . & . & . & . & . \\ a_{j1} & a_{j2} & . & . & . & a_{jn} \\ . & . & . & . & . & . \\ a_{m1} & a_{m2} & . & . & . & a_{mn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & . & . & . & a_{1n} \\ a_{21} & a_{22} & . & . & . & a_{2n} \\ . & . & . & . & . & . \\ a_{j1} & a_{j2} & . & . & . & a_{jn} \\ . & . & . & . & . & . \\ a_{i1} & a_{i2} & . & . & . & a_{in} \\ . & . & . & . & . & . \\ a_{m1} & a_{m2} & . & . & . & a_{mn} \end{bmatrix}$$

Similarly,

$$E(i,j)\times b = \begin{bmatrix} | & | & . & | & . & | & . & | \\ e_1 & e_2 & . & e_j & . & e_i & . & e_m \\ | & | & . & | & . & | & . & | \end{bmatrix} \times \begin{bmatrix} b_1 \\ b_2 \\ . \\ b_i \\ . \\ b_j \\ . \\ b_m \end{bmatrix} = b_1e_1+b_2e_2+....+b_ie_j+...+b_je_i+...+b_me_m = \begin{bmatrix} b_1 \\ b_2 \\ . \\ b_j \\ . \\ b_i \\ . \\ b_m \end{bmatrix}$$

Therefore multiplying one the left with $E(i,j)$ exchanges the i-th row with the j-th row and corresponds to the first elementary row operation.

Now we will look at the second elementary row operation. The matrix s given by $T(i,j,c)$ which is the identity matrix with c appearing in the (i,j) position.

$$T(i,j,c) = \begin{bmatrix} 1 & 0 & 0 & . & . & 0 \\ 0 & 1 & 0 & c & . & 0 \\ & & & & & \\ . & . & . & . & . & 0 \\ 0 & 0. & & . & . & 1 \end{bmatrix}_{m\times m} = \begin{bmatrix} | & | & . & . & | \\ e_1 & e_2 & . & . & e_m \\ | & | & . & . & | \end{bmatrix}_{m\times m} + \begin{bmatrix} 0 & 0 & 0 & . & . & 0 \\ 0 & 0 & 0 & c & . & 0 \\ & & & & & \\ . & . & . & . & . & 0 \\ 0 & 0. & & . & . & 0 \end{bmatrix}_{m\times m}$$

On multiplying $t(i,j,c)$ on the left of A,

$$T(i,j,c)\times A = \left(\begin{bmatrix} | & | & . & . & | \\ e_1 & e_2 & . & . & e_m \\ | & | & . & . & | \end{bmatrix}_{m\times m} + \begin{bmatrix} 0 & 0 & 0 & . & . & 0 \\ 0 & 0 & 0 & c & . & 0 \\ & & & & & \\ . & . & . & . & . & 0 \\ 0 & 0. & & . & . & 0 \end{bmatrix}_{m\times m} \right) \times \begin{bmatrix} a_{11} & a_{12} & . & . & . & a_{1n} \\ . & . & . & . & . & . \\ a_{i1} & a_{i2} & . & . & . & a_{in} \\ . & . & . & . & . & . \\ a_{j1} & a_{j2} & . & . & . & a_{jn} \\ a_{m1} & a_{m2} & . & . & . & a_{mn} \end{bmatrix}_{m\times n}$$

$$\left(\begin{bmatrix} | & | & . & . & | \\ e_1 & e_2 & . & . & e_m \\ | & | & . & . & | \end{bmatrix}_{m\times m} \times \begin{bmatrix} a_{11} & a_{12} & . & . & . & a_{1n} \\ . & . & . & . & . & . \\ a_{i1} & a_{i2} & . & . & . & a_{in} \\ . & . & . & . & . & . \\ a_{j1} & a_{j2} & . & . & . & a_{jn} \\ a_{m1} & a_{m2} & . & . & . & a_{mn} \end{bmatrix}_{m\times n} \right)$$

$$+ \left(\begin{bmatrix} 0 & 0 & 0 & . & . & 0 \\ 0 & 0 & 0 & c & . & 0 \\ . & . & . & . & . & 0 \\ 0 & 0. & & . & . & 0 \end{bmatrix}_{m\times m} \times \begin{bmatrix} a_{11} & a_{12} & . & . & . & a_{1n} \\ . & . & . & . & . & . \\ a_{i1} & a_{i2} & . & . & . & a_{in} \\ . & . & . & . & . & . \\ a_{j1} & a_{j2} & . & . & . & a_{jn} \\ a_{m1} & a_{m2} & . & . & . & a_{mn} \end{bmatrix}_{m\times n} \right)$$

$$= \begin{bmatrix} a_{11} & a_{12} & . & . & . & a_{1n} \\ . & . & . & . & . & . \\ a_{i1} & a_{i2} & . & . & . & a_{in} \\ . & . & . & . & . & . \\ a_{j1} & a_{j2} & . & . & . & a_{jn} \\ a_{m1} & a_{m2} & . & . & . & a_{mn} \end{bmatrix}_{m\times n} + \begin{bmatrix} 0 & 0 & . & . & . & 0 \\ . & . & . & . & . & 0 \\ ca_{j1} & ca_{j2} & . & . & . & c_{jn} \\ . & . & . & . & . & . \\ 0 & 0 & . & . & . & 0 \\ 0 & 0 & . & . & . & 0 \end{bmatrix}_{m\times n}$$

$$= \begin{bmatrix} a_{11} & a_{12} & . & . & a_{1n} \\ . & . & . & . & . \\ a_{i1} + ca_{j1} & a_{i2} + ca_{j2} & . & . & a_{in} + ca_{jn} \\ . & . & . & . & . \\ a_{j1} & a_{j2} & . & . & a_{jn} \\ a_{m1} & a_{m2} & . & . & a_{mn} \end{bmatrix}_{m\times n}$$

Similarly we can show that on multiplying on the left to b,

$$T(i,j,c) \times b = \left(\begin{bmatrix} | & | & . & . & | \\ e_1 & e_2 & . & . & e_m \\ | & | & . & . & | \end{bmatrix}_{m \times m} + \begin{bmatrix} 0 & 0 & 0 & . & . & 0 \\ 0 & 0 & 0 & c & . & 0 \\ . & . & . & . & . & 0 \\ 0 & 0. & & . & . & 0 \end{bmatrix}_{m \times m} \right) \times \begin{bmatrix} b_1 \\ b_2 \\ . \\ b_i \\ . \\ b_j \\ . \\ b_m \end{bmatrix}$$

$$\begin{bmatrix} b_1 \\ b_2 \\ . \\ b_i \\ . \\ b_j \\ . \\ b_m \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ . \\ cb_j \\ . \\ 0 \\ . \\ 0 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ . \\ b_i + cb_j \\ . \\ b_j \\ . \\ b_m \end{bmatrix}$$

□

Note that both $E(i,j)$ and $T(i,j,c)$ are invertible matrices. Specifically, the inverse of $E(i,j)$ is $E(i,j)$ and the inverse of $T(i,j,c)$ is $T(i,j,-c)$

Proposition Elementary row operations do not change the solutions of $Ax = b$

Proof. Let $Ax = b$ represent a system of linear equations where $A \in M_{m \times n}(F)$, $b \in F^m$, and $x \in F^n$. Let $C \in M_{m \times m}(F)$ be a matrix corresponding to an elementary row operation. If $x \in F^n$ is a solution to the equation $Ax = b$ then this implies that x is a solution to equation $C \times Ax = C \implies b \iff (CA)x = Cb$ by associativity. Then if x is a solution of $CAx = Cb$

then x is a solution to $C^{-1}CAx = C^{-1}Cb \iff Ax = b$. This is possible since all elementary row matrices have an inverse. Therefore, elementary row operations do not change the solutions of $Ax = b$.

□

Now that we know that elementary row operations do not change the solutions to the system of equations, we want to look at how they must be applied to reduce a system to row echelon form.

Gaussian Elimination Algorithm

The process can be better understood with the help of an example,
Let us look at the system,
$x_1 + 3x_2 + 2x_3 = 1$
$4x_1 + 0x_2 + x_3 = 1$
$3x_1 + 0x_2 + 5x_3 = 1$

This can be represented as

$$\begin{bmatrix} 1 & 3 & 2 \\ 4 & 0 & 1 \\ 3 & 0 & 5 \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

We would like to reduce A to a lower triangular form. A lower triangular matrix A of size m x n, is one such that $\forall i \in \{1, ...m\}$ and $\forall i \in \{1, ...n\}$, $i < j \rightarrow A_{ij} = 0$.

- The aim is reduce every A_{ij} where $i < j$ starting from the right most column.
- We begin by looking at the A_{mn} element. Check if it is non-zero, in our example, it is 5 so we proceed.

- Let c = -A_{1n}/A_{mn} = -2/5. Multiply a $T(-2/5, 1, n)$ with c in the 1st row , n-th column.

$$\begin{bmatrix} 1 & 0 & -2/5 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 3 & 2 \\ 4 & 0 & 1 \\ 3 & 0 & 5 \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & -2/5 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

$$\iff \begin{bmatrix} -1/5 & 3 & 0 \\ 4 & 0 & 1 \\ 3 & 0 & 5 \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3/5 \\ 1 \\ 1 \end{bmatrix}$$

- Now similarly we take $c = -A_{2n}/A_{mn} = -1/5$. Multiply with $T(-1/5, 2, n)$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -1/5 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} -1/5 & 3 & 0 \\ 4 & 0 & 1 \\ 3 & 0 & 5 \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -1/5 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 3/5 \\ 1 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} -1/5 & 3 & 0 \\ 17/5 & 0 & 0 \\ 3 & 0 & 5 \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3/5 \\ 4/5 \\ 1 \end{bmatrix}$$

- We have now cleared the necessary elements in the last column, so we go to the second last one, 2nd in this case. Look at the A_{22} element. Since, it is zero we cannot create a c using it , so we look if there is any other non-zero element in the column that needs to become zero. The A_{12} element is such. So we do a row swap of the first and second row.

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} -1/5 & 3 & 0 \\ 17/5 & 0 & 0 \\ 3 & 0 & 5 \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 3/5 \\ 4/5 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} 17/5 & 0 & 0 \\ -1/5 & 3 & 0 \\ 3 & 0 & 5 \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 4/5 \\ 3/5 \\ 1 \end{bmatrix}$$

- We now have a lower triangular matrix. If this was not done, we would repeat the procedure from before finding appropriate c and $T(i, j, c)$, and clearing out each column in order. If at any point you cannot find a c because of the presence of a zero you multiply the appropriate $E(i, j)$.

Once you have completed the procedure, you do back substitution to find the values of x. Let $Cx = d$ be the new system in row echelon form. The back substitution algorithm works as follows:

$$x_1 = \frac{d_1}{C_{11}}$$

$$x_i = \left(d_i - \sum_{k=1}^{i} A_{ik} x_k \right) / A_{ii}$$

If the denominator for any of the divisions is zero, look at the numerator. If that is also zero you have infinitely many solutions, otherwise, you have no solutions.

Looking at our example,

$$x_1 = \frac{4/5}{17/5} = 4/17$$

$$x_2 = \frac{3/5 - (-4/85)}{3} = 17/85$$

$$x_3 = \frac{1 - (12/17)}{5} = 1/17$$

This is the process of solving a system of linear equations using Gaussian Elimination.

Chapter 6

Formalising Gaussian Elimination

Within the Mathlib library in Lean, there is an extensive collection of undergraduate mathematics that has been formalised. In addition to this, the different topics that are yet to be done in Lean's Mathlib library have been specified on the community webpage. One such topic that is yet to be done is Gaussian Elimination. Through our project, we attempted to formalise the process of Gaussian elimination. This was done utilising the already existing definitions, theorems and proofs available in Mathlib library, specifically in the Linear Algebra folder.

Linear Algebra in Mathlib

Gaussian Elimination is a topic listed under linear algebra within the Mathlib library. Therefore, in order to understand how to go about formalising the process of Gaussian elimination, our first objective was to go through all the concepts in Linear Algebra that have been already formalised in the library.
On studying the library we found out that the following are the main topics of undergraduate-level Linear Algebra that exists in Mathlib.

- Under the fundamentals, the topics surrounding modules, linear maps, the category of modules over a ring, vector spaces, quotient spaces, tensor product, noetherian modules, basis, multilinear map, alternating map, and general linear groups have been formalised.

- Regarding finite-dimensional vector spaces, the main ideas that exist in the library are about finite dimensionality, isomorphism with K^n, and isomorphism with bidual.
- Under duality, the concepts of dual vector spaces and dual basis exist in the library.
- The structure theorem for finitely generated modules in a PID has been done.
- Under the main section of matrices, the ideas of ring-valued matrix, matrix representation of a linear map, determinant and invertibility have been done.
- On the topic Endomorphism polynomials the concepts regarding minimal polynomial, characteristic polynomial and the Cayley-Hamilton theorem have been formalised.
- On the Structure theory of endomorphisms, the concepts pertaining to eigenvalue, eigenvector, existence of an eigenvalue has been done
- In the topic Bilinear and quadratic forms, the formalised sections include bilinear form, alternating bilinear form, symmetric bilinear form, matrix representation, quadratic form and polar form of a quadratic.
- Lastly under Finite-dimensional inner product spaces the concepts of existence of orthonormal basis, diagonalization of self-adjoint and endomorphisms has been done.

Mathlib.Matrix.Transvection

While studying the Linear Algebra folder of the Mathlib Library, the part that came closest to what we wanted to achieve was the file Mathlib.Matrix.Transvection. Within this file, the second row operation that is required for Gaussian elimination has been defined in addition to various properties and theorems regarding it. We also found that much of the structure of this file could be followed while writing our proof for Gaussian elimination.

The file begins by defining what is a transvection matrix. The definition takes three parameters i j and c that specify that c will be in the ith row jth column of the identity matrix.This is the

same as the $T(i.j, c)$ matrix defined in the previous chapter. The first section proves different simple theorems about transvection matrices. These include

- Multiplication by a transvection i j c on the left of a matrix M is the same as updating the ith row with c times the jth row.
- Multiplication by a transvection i j c on the right of a matrix M is the same as updating the jth column with c times the ith column.
- Multiplication of two transvection matrices would result in the following := transvection i j c * transvection i j d = transvection i j (c + d)
- The determinant of a transvection matrix is 1.

The second section defines transvection as a structure, as was mentioned in the 3rd chapter. Defining it as a structure brings the ease of having direct access to all the relevant fields. A function is defined that takes a transvection structure as an input and converts it to a matrix. Certain theorems are stated that prove that this conversion is consistent. It is also shown that creating a list of transvection structures applying the conversion to a matrix and taking the product will give the correct output.

In Lean, n⊕p is the disjoint union of types n and p(can be viewed as sets in this context). The next part of the code defines a new function on transvection structure. It converts a transvection structure having n as the indexing set for the corresponding matrix to a transvection structure having n ⊕ p as the indexing set for the corresponding matrix. Defining such a function provides the ability to extract the different blocks of a given matrix. For example, a square matrix M having n⊕p as the indexing set can be viewed as
M = $\begin{bmatrix} \begin{bmatrix} M_{11} \end{bmatrix}_{nxn} & \begin{bmatrix} M_{12} \end{bmatrix}_{nxp} \\ \begin{bmatrix} M_{21} \end{bmatrix}_{pxn} & \begin{bmatrix} M_{22} \end{bmatrix}_{pxp} \end{bmatrix}$ A theorem has been proved explicitly showing that.
With this the set up has been complete to prove the main theorem in the file.

The section three proves the following theorem - Let M be a n x n square matrix. There exists two list of transvections $L = [L_1, L_2, ..., L_i]$ and $R = [R_1, R_2....R_j]$ such that $L_i.....L_2 \times L_1 \times M \times R_1 \times R_2....R_j = D$, where D is some diagonal matrix.

In words, for every square matrix, can be reduced to diagonal form by left and right multiplication by transvections. Specifically, there exists two lists of transvections such that multiplying the matrix on the left and right with these lists will result in a diagonal matrix.

Certain definitions have been stated and intermediary theorems have been proved in this section. The important ones are :

- **Definition** (listTransvecCol M) - A list of transvections $L = [L_1, L_2, ..., L_i]$ such that $M' = L_i.....L_2 * L_1 * M$ and $M_{i,n} = 0, \forall i \in \{1, 2, ...n-1\}$
- **Definition** (listTransvecRow M) - A list of transvections $R = [R_1, R_2, ..., R_j]$ such that $M' = M \times R_1 \times R_2... \times R_i$ and $M_{n,i} = 0, \forall i \in \{1, 2, ...n-1\}$
- **Theorem** - Let $M_{nn} \neq 0$ and listTransvecRow M $= R = [R_1, R_2, ..., R_j]$, then $M' = M \times R_1 \times R_2... \times R_i$ and $M_{n,i} = 0, \forall i \in \{1, 2, ...n-1\}$. This proves that the list that is defined does precisely what we expect it to do.
- **Theorem** - Let $M_{nn} \neq 0$ and listTransvecCol M $= L = [L_1, L_2, ..., L_i]$, then $M' = L_i.....L_2 \times L_1 \times M$ and $M_{i,n} = 0, \forall i \in \{1, 2, ...n-1\}$. This proves that the list that is defined does precisely what we expect it to do.
- **Theorem** - Let $M_{nn} \neq 0$ and listTransvecRow M $= [R_1, R_2, ..., R_j]$ and listTransvecCol M $= L = [L_1, L_2, ..., L_i]$. Then M' $= L_i.....L_2 \times L_1 \times M \times R_1 \times R_2....R_j$ is of the form M' $= \begin{bmatrix} \begin{bmatrix} M'' \end{bmatrix} & 0 \\ 0 & M_{nn} \end{bmatrix}$. Here M" is an (n-1) x (n-1) matrix. M' is a block-diagonal matrix.
- **Theorem** - There exist two lists of 'TransvectionStruct' such that multiplying by them on the left and on the right makes a matrix block-diagonal.

- The next theorem is the induction step for the reduction.

  Fin r = {0,1,2...r-1} where $r \in \mathbb{N}$

  Unit is a set with a single element.

  Theorem - Let M be a ((r+1) x (r+1)) matrix indexed by the set Fin r$\oplus$Unit. For all N such that N is a (r x r) matrix indexed by the set Fin r, there exists a two list of transvections R = $[R_1, R_2, ..., R_j]$ and $L = [L_1, L_2, ..., L_i]$ such that $L_i.....L_2 \times L_1 \times M \times R_1 \times R_2....R_j = D$, where D is some diagonal matrix. Then M can be reduced by transvections into a diagonal matrix D'.

- Using induction, it is finally proved that,

  Theorem - Any matrix can be reduced to diagonal form by elementary operations.

- As a corollary,

  Theorem - Any matrix can be written as the product of transvections, a diagonal matrix, and transvections.

Chapter 7

Row Exchange Matrix

In the first section of our code, we define a row exchange matrix that corresponds to the elementary operation of exchanging two rows. This matrix is the same as $E(i, j)$ in chapter 5. We also state and prove various theorems corresponding to row exchange matrices.

We begin by defining two variables -

- n will be used as the index when defining a matrix. We define a Fintype structure on n making it a finite type. The DecidableEq structure asserts that n has decidable equality, that is, a = b is decidable for all a b : n
- The entries of the matrix will be of type R. R has a commutative ring structure on it

```
variable {n : Type*} [DecidableEq n] [Fintype n]
variable {R : Type v} [CommRing R]
```

The first is to define a matrix corresponding to the elementary row operation of exchanging rows. n x n Identity matrices with the ith and jth row swapped is defined by RowEx i j. We are utilizing the already existing method of defining permutation matrices to define our RowEx matrix.

Note that we define a matrix using the following function that is already in Mathlib.

```
def Matrix (m : Type u) (n : Type u') (R : Type v) :
Type (max u u' v)
```

`Matrix m n R` - defines a matrix whose rows are indexed by m, columns are indexed by n and the entries are from R

So we define our row exchange matrix as,

```
def RowEx (i j : n): Matrix n n R :=
(Equiv.swap i j).toPEquiv.toMatrix
```

There are a few important theorems and properties that already exist in Lean's library that are being utilised to prove results in this section.

To begin with,

```
theorem PEquiv.equiv_toPEquiv_toMatrix {n : Type u_4} {α : Type v} [DecidableEq n]
    [Zero α] [One α] (σ : n ≃ n) (i : n) (j : n) : PEquiv.toMatrix (Equiv.toPEquiv σ
    ) i j = 1 (σ i) j
```

This theorem gives us that a permutation matrix can be defined as the matrix achieved by permuting the rows of the identity matrix. Once you have your RowEx matrix in this form, it becomes easier to work with it.

The next theorem that is being used is as follows. It tells you what the output would be in different cases when your `Equiv.swap` function is evaluated at a particular x.

```
theorem Equiv.swap_apply_def.{u_1} {α : Sort u_1} [inst : DecidableEq α] (a b x : α)
    : ↑(Equiv.swap a b) x = if x = a then b else if x = b then a else x
```

A consequence of this definition is, `Equiv.swap_comm` - Equiv.swap a b = Equiv.swap b a

Another important set of theorems we require are those regarding updating rows of a matrix. The following theorem says that when you update the i-th row of a matrix M with a row matrix k, the M_{ab} element of the matrix will be k_b is $i = a$ and M_{ab} otherwise.

```
theorem Matrix.updateRow_apply {m : Type u_2} {n : Type u_3} {α : Type v} {M : Matrix
    m n α} {i : m} {j : n}{b : n → α} [inst : DecidableEq m] {i' : m} : updateRow M
    i b i' j = if i' = i then b j else M i' j
```

When you update the i-th row of a matrix M with a row matrix k, the i-th row of the matrix will be equal to k.

```
theorem Matrix.updateRow_self {m : Type u_2} {n : Type u_3} {α : Type v} {M : Matrix
    m n α} {i : m} {b : n → α} [inst : DecidableEq m] : updateRow M i b i = b
```

When observing all the theorems listed up until now, most of them have conditionals and if statements as a part of their definition. These also include an equality or inequality as the conditions. The next three theorems assist in simplifying these.

`if_pos` - This is used when the conditional in a statement holds true.

`if_neg` - This is used when the conditional in a statement is false

`ne_comm` - $a \neq b \to b \neq a$

With this setup we can start proving our own theorems. Most of them are simple ones using already existing properties. The code for all the theorems we proved will be listed in the appendix. Over here, we will be looking at and understanding the proof technique for one of them.

The theorem states that RowEx i j represents a matrix that is precisely swapping the ith row of the identity matrix with the jth one and swapping the jth row of the identity row with the ith one.

Proving this theorem provides a definition for Rowex i j that makes computation simpler.

```
theorem updaterow_eq_swap [Finite n]:
updateRow (updateRow (1 : Matrix n n R) i ((1 :Matrix n n R) j)) j ((1 : Matrix n n
    R) i) = RowEx i j := by
--The following command allows us to look and compare each specific element of the
    matrices on both sides of the equality
ext a b
--We consider the different cases where the element that we are considering are in
    the i or j row and when it is not. Then the simplification is done accordingly.
```

```
by_cases ha : i = a; by_cases hb : j = b
· simp[ha,hb]
  rw[RowEx,PEquiv.equiv_toPEquiv_toMatrix,Equiv.swap_apply_left]
  rw[Matrix.updateRow_apply,Matrix.updateRow_self]
  by_cases hab : a = b
  · simp[hab]
  · rw[if_neg hab]
· rw[ha,RowEx]
  rw[PEquiv.equiv_toPEquiv_toMatrix,Equiv.swap_apply_left]
  rw[Matrix.updateRow_apply,Matrix.updateRow_self]
  by_cases haj : a = j
  · rw[haj,if_pos rfl]
  · rw[if_neg haj]
· rw[RowEx]
  rw[PEquiv.equiv_toPEquiv_toMatrix,Matrix.updateRow_apply]
  rw[Matrix.updateRow_apply,Equiv.swap_apply_def]
  by_cases haj : a = j
  · rw[if_pos haj,if_neg (ne_comm.mp ha),if_pos haj]
  · simp[if_neg haj,if_neg (ne_comm.mp ha)]
```

Let us observe how the goals in the Lean info view change with each line of code for the first of our cases. The rest of the cases follow a similar pattern.

``ha : i = a hb : j = b ⊢ updateRow (updateRow 1 i (OfNat.ofNat 1 j)) j (OfNat.ofNat i) a b = RowEx i j a b```	```ha : i = a hb : j = b ⊢ updateRow (updateRow 1 a (OfNat.ofNat 1 b)) b (OfNat.ofNat 1 a) a b = RowEx a b a b```	```ha : i = a hb : j = b ⊢ (if a = b then OfNat.ofNat 1 a b else OfNat.ofNat 1 b b) = OfNat.ofNat 1 b b```
Let RowEx i j be the row exchange matrix M. In this case, we are considering an element $M_{a,b}$ where a = i and b = j.	In line 3, we use simp to substitute the value i and j as a and b respectively	In line 4, we simplify the goal using all the theorems specifies to obtain this new goal.
``ha : i = a hb : j = b hab : a = b ⊢ (if a = b then OfNat.ofNat 1 a b else OfNat.ofNat 1 b b) = OfNat.ofNat 1 b b```	```ha : i = a hb : j = b hab : ¬a = b ⊢ (if a = b then OfNat.ofNat 1 a b else OfNat.ofNat 1 b b) = OfNat.ofNat 1 b b```	And with that we have completed the proof for the first case. The other cases use a similar strategy.
We consider two cases of $a = b$ and $a \neq b$. Here using `if_pos` we simplify and solve the goal of $a = b$	When $a \neq b$ we use `if_neg` and close the goal	

The other theorems that have been proved about Row Exchange matrices are the following

-

- `theorem RowExmul_eq_swap (i j: n)(M : Matrix n n R) : (RowEx i j : Matrix n n R) * M = updateRow (updateRow (M) i (M j)) j (M i) :=`

  Multiplying with a matrix M with RowEx i j on the left exchanges the ith row and the jth row of M with each other

- RowEx i j = RowEx j i, giving that the operation is commutative.

- RowEx i j and RowEx j i are inverses of each other

- RowEx i j is the inverse of itself

- The determinant of RowEx i j when i j is -1

Chapter 8

Elimination Matrices

In the chapter 5 in the process of Gaussian Elimination we saw that, in order to clear out the elements of a given column, we require a single row exchange and then multiple transvections.

An Elimination Matrix for a n x n matrix M is defined to be a matrix E such that M' = E*M and $M'_{i,n} = 0, \forall 1 \leq i \leq n-1$.

As done in the formalisation of transvections we would also like to create a structure representing elimination matrices called ElimStruct . We define an elimination matrix to be of the form $E = L_i \times ..... \times L_1 \times R$ where L_i's are transvections and R is a row exchange matrix.

In the next chapter it will be shown that this is precisely the matrix that is required to clear out a column.And ultimately we would like to show that applying a list of Elim matrices will reduce any given matrix into lower triangular form.

The structure `elimStruct` stores the information of three things within it. Namely, the list of transvection structures corresponding to the transvection matrices and i j which are the input for the RowEx matrix .

```
structure elimStruct where
(L : List (TransvectionStruct n R))
(i j: n)
```

The toElim function takes an elimStruct to its matrix form

```
def toElim (e : elimStruct n R) : Matrix n n R :=
((e.L).map toMatrix).prod \times (RowEx e.i e.j)
```

In the next step we would like to be able to convert elimStruct n R to elimStruct n $\oplus$ p R as seen in the transvection file.

To be able to do this we have proved the following theorem about RowEx matrices. The code for this have been provided in the appendix.

Theorem - Let M an n x n matrix and R be a row exchange matrix of dimension n x n. Then

$$\begin{bmatrix} \begin{bmatrix} R \times M \end{bmatrix} & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} R \end{bmatrix} & 0 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} \begin{bmatrix} M \end{bmatrix} & 0 \\ 0 & 1 \end{bmatrix}$$

Note - The fromBlocks function forms a single large matrix by flattening smaller 'block' matrices of compatible dimensions.

```
theorem rowExInl (M: Matrix (Fin r) (Fin r) k) (i j :Fin r) : fromBlocks ((RowEx i
    j)*M) 0 0 (1: Matrix Unit Unit k) = (RowEx (inl i) (inl j))* (fromBlocks M 0 0
    (1: Matrix Unit Unit k)) :=
```

For the purpose of Gaussian Elimination, we are taking our matrices to have the indexing type Fin r which is natural numbers from 0 to r-1. And the entries in the matrix will be of type k where k is a field.

```
variable (k :Type) [Field k]
```

Then as done in the transvection code, we define a function that takes an n indexed elimStruct to an n $\oplus$ p indexed elimStruct . As we said before structure elimStruct stores the information of three things within it. Namely, the list of transvection structures corresponding to the transvection matrices and i j which are the input for the RowEx matrix

```
def elimSum_Inl (e : elimStruct n R ) : (elimStruct (n ⊕ p) R ) where
L := ((e.L).map (sumInl p))
i := inl e.i
j := inl e.j
```

Over here the inl command does the following. We require i and j to be of type Fin r $\oplus$ Unit. Since we are defining it to be the same i and j as in our elimStruct of index Fin r, the inl command creates an object of type Fin r $\oplus$ Unit taking an object of type Fin r.

The theorem shows that using the elimSum_Inl function is equal to applying fromBlocks as written in the theorem.

```
theorem toMat_sumInl (e : elimStruct (Fin r) k) : (e.elimSum_Inl).toElim = fromBlocks
    e.toElim 0 0 (1 : Matrix Unit Unit k) :=
```

The same holds as above even for a list of elimStructs. This has been proved using induction.

```
theorem list_Elim_prod_eq (M : Matrix (Fin r) (Fin r) k) (L : List (elimStruct (Fin
    r) k)) (N : Matrix Unit Unit k) (O : Matrix Unit (Fin r) k): (L.map (toElim ∘
    elimSum_Inl)).prod * fromBlocks M (0: Matrix (Fin r) Unit k) O N = fromBlocks
    (L.map toElim).prod * M) (0: Matrix (Fin r) Unit k) O N :=
 induction' L with t L IH
 · simp
 · simp[Matrix.mul_assoc, IH, toMat_sumInl, fromBlocks_multiply]
```

. Let us look at this proof in the lean info view.

We use induction to prove this.

- The base case looks at an empty list [ ]. This case can be easily simplified using simp.

```
⊢ List.prod (List.map (toElim ∘
elimSum_Inl) []) * fromBlocks M 0 0
N =
  fromBlocks (List.prod (List.map
toElim []) * M) 0 0 N
```

- The induction step assumes that the equality holds for some list L. The induction step is to prove that it will hold for a new list L'= L++t. That is, when some element t is added to the list L.

```
IH : List.prod (List.map (toElim ∘
elimSum_Inl) L) * fromBlocks M 0 0 N
=
  fromBlocks (List.prod (List.map
toElim L) * M) 0 0 N
⊢ List.prod (List.map (toElim ∘
elimSum_Inl) (t :: L)) * fromBlocks
M 0 0 N =
  fromBlocks (List.prod (List.map
toElim (t :: L)) * M) 0 0 N
```

- The induction step can be proved using the previous theorem and some basic properties of matrices.

The main gist of this section is to provide the following: Let E be a Elim matrix where $E = L_i \times ..... \times L_1 \times R$ where L_i's are transvections and R is a row exchange matrix. Then,

E' = $\begin{bmatrix} \begin{bmatrix} L_i \end{bmatrix} & 0 \\ 0 & 1 \end{bmatrix} \times ..... \times \begin{bmatrix} \begin{bmatrix} L_1 \end{bmatrix} & 0 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} \begin{bmatrix} R \end{bmatrix} & 0 \\ 0 & 1 \end{bmatrix}$ is an Elim Matrix.

Chapter 9

Gaussian Elimination

In this chapter we show that any square matrix can be reduced to a lower triangular matrix through elementary row operations. This is the same as showing that any matrix can be reduced to a lower triangular matrix by left of transvection and row exchange matrices. This is the result of Gaussian Elimination.

The first theorem in this section states the following,

Theorem - For every matrix M indexed by Fin $\oplus$ Unit (It will be an (r+1) x (r+1) matrix), there is a list of transvections and a RowEx matrix such that multiplying on the left with the RowEx and then the list of transvections will make $M_{(i,r+1)} = 0$ for every $1 \leq i < r + 1$.

```
theorem transvec_RowEx_mul_lastcol (M : Matrix (Sum (Fin r) Unit) (Sum (Fin r) Unit)
    k) :  ∃ i :Fin r ⊕ Unit, ∃ L : List (TransvectionStruct (Sum (Fin r) Unit) k), (∀
    j : Fin r,  ((L.map toMatrix).prod *(((RowEx i (inr unit) : Matrix (Sum (Fin r)
    Unit) (Sum (Fin r) Unit) k)) * M)) (inl j) (inr unit) = 0) := by
```

The first step of our proof is to divide the goal into two cases. One where $M_{(r+1,r+1)}$ is zero and the other when it is non-zero.

```
 by_cases M (inr unit) (inr unit) ≠ 0
```

Once that was been done we solve for the first case where $M_{(r+1,r+1)}$ is non-zero

We begin by creating the i and L that is required and inserting it in the goal. Here we need

every value from 0 to r, so we are giving it the indexing set Fin r. Since the corner element is already non-zero we can define our list L to be a RowEx Matrix that is identity and the rest of the elements to be the list of transvections required to clear out the last column.

```
·let a : Fin r ⊕ Unit := inr unit
 exists a
 let L : List (TransvectionStruct (Sum (Fin r) Unit) k) :=
  List.ofFn fun i : Fin r =>
    ⟨inl i, inr unit, by simp, - (((RowEx a (inr unit) : Matrix (Sum (Fin r) Unit)
  (Sum (Fin r) Unit) k)) * M) (inl i) (inr unit) /
     (((RowEx a (inr unit) : Matrix (Sum (Fin r) Unit) (Sum (Fin r) Unit) k)) * M)
  (inr unit) (inr unit)⟩
 refine' ⟨ L,_⟩
 intro j
```

We simplify our goal using properties of transvections and RowEx matrices and prove the required property.

```
have A : L.map toMatrix = listTransvecCol (((RowEx a (inr unit) : Matrix (Sum (Fin
  r) Unit) (Sum (Fin r) Unit) k)) * M)
 := by simp [listTransvecCol, (· ∘ ·)]
 rw[A]
 simp[RowExid]
 rw[listTransvecCol_mul_last_col]
 exact h
```

Now we look at the second case. Over here $M_{(r+1,r+1)} = 0$.

```
· push_neg at h
```

Within the second case we consider two cases. The first one being when atleast one entry in last column is non-zero and the second one being when all entries are zero

```
  by_cases (∃ i : Fin r, M (inl i) (inr unit) ≠ 0)
  · cases h with
    |intro i hi =>
```

In part one of the second case we have that atleast one entry in the last column is non-zero. Using a row exchange we can make the $M_(r+1, r+1) \neq 0$.

```
  · have hn : (((RowEx (inl i) (inr unit) : Matrix (Sum (Fin r) Unit) (Sum (Fin r)
Unit) k) * M) (inr unit) (inr unit) ≠ 0) := by
    rw[RowExmul_eq_swap]
    rw[Matrix.updateRow_self]
    exact hi
```

Once that is done we can repeat a proof similar to Case 1 since $M_{(r+1,r+1)}$ is non-zero

```
    let a : Fin r ⊕ Unit := inl i
    exists a
    let L : List (TransvectionStruct (Sum (Fin r) Unit) k) :=
    List.ofFn fun i : Fin r =>
    ⟨inl i, inr unit, by simp, - (((RowEx a (inr unit) : Matrix (Sum (Fin r)
Unit) (Sum (Fin r) Unit) k)) * M) (inl i) (inr unit) /
    (((RowEx a (inr unit) : Matrix (Sum (Fin r) Unit) (Sum (Fin r) Unit) k)) * M)
(inr unit) (inr unit)⟩
    refine' ⟨ L,_⟩
    intro j
    have A : L.map toMatrix = listTransvecCol (((RowEx a (inr unit) : Matrix (Sum
(Fin r) Unit) (Sum (Fin r) Unit) k)) * M)
    := by simp [listTransvecCol, (· ∘ ·)]
    rw[A]
    rw[listTransvecCol_mul_last_col]
    exact hn
```

Now we look at the second part of the second case. In this case, since our entire column is already completely zeroes, we are not required to reduce anything. So, we choose our transvections and RowEx matrix to be the identity matrix. And the simplification is done accordingly.

```
·push_neg at h
let a : Fin r ⊕ Unit := inr unit
```

```
exists a
let L : List (TransvectionStruct (Sum (Fin r) Unit) k) :=
List.ofFn fun i : Fin r =>
⟨inl i, inr unit, by simp, 0 ⟩
exists L
intro j
have A : L.map toMatrix = listid r := by
 simp[listid, (· ∘ ·)]
rw[A,listid_prod_eq_id]
simp[h,RowExid]
```

The next theorem we have proved is the same theorem as above in terms of elim matrices.

```
theorem exists_elimmatrix_mul_lastcol (M : Matrix (Sum (Fin r) Unit) (Sum (Fin r)
    Unit) k) : ∃(N : elimStruct (Fin r ⊕ Unit) k), (∀ j : Fin r , ((N.toElim) * M)
    (inl j) (inr unit) = 0) :=
```

Now we prove the inductive step for our final theorem. If for all r x r matrices, there exists a list of Elimination matrices that reduce them to lower triangular matrices, then given a (r + 1) x (r + 1) matrix, there exists a list of Elim matrices that reduce it to a lower triangular matrix.

```
theorem exists_ListElimMatrix_eq_UpperTriangular(IH : ∀ (M : Matrix (Fin r) (Fin
r) k), ∃ ( E :List (elimStruct (Fin r) k))  , ((E.map toElim).prod *
M).BlockTriangular OrderDual.toDual) (M :Matrix (Sum (Fin r) Unit) (Sum (Fin r)
Unit) k) : ∃ (E₁ : List (elimStruct (Fin r ⊕ Unit) k) ), ((E₁.map toElim).prod *
M).BlockTriangular OrderDual.toDual := by
```

Using the theorem `exists_elimmatrix_mul_lastcol` we can show that there an Elim matrix that clears out the last column of our r+1 by r+1 matrix M. Let this be N.

```
have HM : ∃ N : elimStruct (Fin r ⊕ Unit) k, ∀ (j : Fin r), (toElim N * M) (inl j)
 (inr Unit.unit) = 0 := by
apply exists_elimmatrix_mul_lastcol M
```

Using the cases tactic we can split the hypothesis HM into an Elim matrix N and the proof that N clears out the last row of M. And we define M' = N*M

```
cases HM with
|intro N HM =>
let M' := N.toElim*M
```

M" is the top left block of M' which is a square matrix indexed by Fin r.

```
let M'' := toBlocks₁₁ M'
```

By applying the induction hypothesis on M" we obtain a list of Elim matrices L and the hypothesis. This list has matrices indexed by Fin r .

```
rcases IH M'' with ⟨L, h₀⟩
set Mₐ := toBlocks₂₁ M'
set c := toBlocks₂₂ M'
```

The next step is to create a list. This list includes list L converted to a list of Fin r ⊕ Unit indexed elim matrices as we defined in chapter 7 and N. N clears out the last column of M, and the list of elim matrices reduces the rest of the matrix. We say the list defined as such is the required list list. After substituting our created list, the goal can be closed using simple tactics

```
refine'⟨L.map (elimSum_Inl) ++ [N],_⟩
suffices ((L.map (toElim ∘ elimSum_Inl)).prod * M').BlockTriangular
  OrderDual.toDual by simpa[Matrix.mul_assoc]
· have H : M' = fromBlocks (M'') 0 Mₐ c := by
    simp
    have X : toBlocks₁₂ (M') = 0 := by
     ext a b
     simp[toBlocks₁₂]
     rw[HM a]
    rw[←X]
    simp[Matrix.fromBlocks_toBlocks]
   rw[H,list_Elim_prod_eq]
   simpa[BlockTriangular]
```

Now we have reached our final theorem,

```
theorem gauss_elim (M' : Matrix (Fin r) (Fin r) k) : ∃ ( E :List (elimStruct (Fin r)
    k))
 , ((E.map toElim).prod * M').BlockTriangular OrderDual.toDual := by
```

This theorem is proved by induction. The base case can be done through utilizing the simp tactic. We have already proved the induction step in the previous theorem. Putting together the two, we complete the process of gaussian elimination.

The final theorem for gaussian eliminations has therefore been proved. It states that given any square matrix M, it can be reduced to a lower triangular matrix through elementary row operations.

Chapter 10

Conclusion

Lean proof assistant is a versatile programming language that has gained popularity in academia as a powerful tool and expressive language that provides a platform to do formal verification. It was developed by Microsoft research under Leanardo de Moura. It has developed over the past few years as an open-source project.

Formal Verification is the use of logical and computational methods to prove mathematical claims. Recent times have seen an increase in the use of computers to do formal verification. This is in the form of automated and interactive theorem provers. Lean combines properties of both automated and interactive theorem provers. It provides an automated system of tactics and proofs that involve user interaction at every step.Mathlib is the mathematics library in Lean. It was developed as a collaborative effort by the Lean community. The Xena Project was introduced as an effort to contribute to formalizing undergraduate mathematics in the Mathlib library.

Lean functions on the underlying notion of type theory. Type theory is a system for categorizing and working with objects based on their types. In Lean, every object can be expressed as a type. Lean role as proof assistant works primarily on a type-checking mechanism. Within Lean's framework there are commands that allow declarations of constants, variables, functions, and

theorems. A theorem is proved by creating an object of the same type as the theorem statement.

Theorem proving in Lean can be assisted by the tactic mode. This mode provides instructions and commands on how to construct a proof. It also breaks down a proof into smaller goals and provides a visual representation of each case through the Lean Infoview.

The parts of undergraduate mathematics that are yet to be formalised have been listed on the documentation page of Mathlib. One such topic is Gaussian elimination in Linear Algebra. Gaussian elimination is the process of using elementary row operations to reduce a square matrix into an upper or lower triangular matrix. The row operation corresponding to multiplying a row by a scalar and adding it to another row has already been formalised within the Mathlib library in the file Transvection.lean. The matrix corresponding to doing this operation is the transvection matrix. Following a similar structure to this file, we were able to formalise the process of Gaussian Elimination.

The first part of the code defines a matrix corresponding to the other row operation which is the exchange of rows. We call this matrix a row exchange matrix. The product of a list of transvections and a row exchange matrix provides something known as an Elimination Matrix. We prove that given a square matrix M, there exists an elimination matrix E such that left multiplication by E on M will clear out all entries in the last column except the last one.

Following this we prove that if a matrix of size $n \times n$ can be reduced to a lower triangular form by a list of elimination matrices, then a matrix of size $(n+1) \times (n+1)$ can be reduced to a lower triangular form by a list of elimination matrices. This is the inductive step for our final theorem. The final theorem states that any square matrix can be reduced to a lower triangular form by elementary row operations.

Engaging with Mathlib and theorem proving in Lean provides an invaluable opportunity to

fortify one's foundational mathematical concepts. Lean's rigorous type-checking mechanism fosters a deeper understanding of mathematical principles and techniques. This was an important takeaway of the project.

In the realm of future research, a direct progression would involve extending the current proof methodology to tackle solving systems of linear equations within the Lean framework. Beyond this, many more mathematical concepts are yet to be formalised in the Mathlib library. Therefore, there is a great opportunity to work on the formalization of these concepts and contribute to the Mathlib community.

Bibliography

Mathlib 4 Documentation (n.d.). URL: https://leanprover-community.github.io/mathlib4_docs/index.html.

Mathematics in Lean (n.d.). URL: https://leanprover-community.github.io/mathematics_in_lean/.

Jeremy Avigad Leonardo de Moura, Soonho Kong and Sebastian Ullrich (n.d.). *Theorem Proving in Lean*. URL: https://lean-lang.org/theorem_proving_in_lean4/title_page.html.

Xena Project (n.d.). URL: https://www.ma.imperial.ac.uk/~buzzard/xena/.

Vogan, David (2019). "Gaussian Elimination". In: URL: https://math.mit.edu/~dav/gauss19.pdf.

Microsoft Research (n.d.). URL: https://www.microsoft.com/en-us/research/project/lean/.

Isabelle Proof Assistant (n.d.). URL: https://isabelle.in.tum.de/.

Coq Proof Assistant (n.d.). URL: https://coq.inria.fr/.

Geo-Gebra (n.d.). URL: https://www.geogebra.org/.

Appendix A

Appendix

A.1 Working on Lean

- Install VS Code from https://code.visualstudio.com/download .It is the recommended editor for Lean
- Open VS Code, go to the "activity bar" along the left-hand side of the screen, and click the "extension" icon. In the search box that appears, type lean4, and then select the lean4 extension that appears, and click the install button.
- Under the File menu, select New text file. A new window labelled Untitled-1 will appear.
- There will be a prompt in this window saying Select a language, which you should click on and select Lean4.
- Once you've set the language to Lean4, a dialog will appear in the bottom right of your screen, saying Failed to start 'lean' language server with a button Install Lean using Elan. Click this button, and inside a terminal window within VS Code you should see the installation process begin. It will take on the order of a minute to download and install Lean.
- When this finishes, return to the Untitled-1 editor, and type

  ```
  #eval 18 + 19.
  ```

  If you see a blue underline appear under #eval, and the result 37 displayed in the right-hand side Lean info view panel, then you have successfully installed Lean 4!
- More information about setting up projects and working on Lean can be found here

A.2 Lean Code

```
import Mathlib.Tactic
import Mathlib.Data.Matrix.Basis
import Mathlib.Data.Matrix.DMatrix
import Mathlib.LinearAlgebra.Matrix.Determinant
```

```
import Mathlib.LinearAlgebra.Matrix.Reindex
import Mathlib.Tactic.FieldSimp
import Mathlib.Data.Real.Basic
import Mathlib.Algebra.BigOperators.Basic
import Mathlib.Data.PEquiv
import Mathlib.LinearAlgebra.Matrix.Transvection
import Mathlib.Logic.Equiv.Basic
import Mathlib.Data.Matrix.PEquiv
import Mathlib.Data.Matrix.Reflection
import Mathlib.Algebra.Group.OrderSynonym

open Matrix BigOperators
open Equiv Equiv.Perm Finset Function

namespace matrix
open matrix
variable {m n p : Type*} [DecidableEq n] [Fintype n] [DecidableEq m] [Fintype
    m][DecidableEq p]

variable {R : Type v} [CommRing R]

variable {k : Type*} [Field k]

--nxn Identity matrices with the ith and jth row swapped is defined by RowEx i j.

def RowEx (i j : n): Matrix n n R :=
(Equiv.swap i j).toPEquiv.toMatrix

--RowEx i i is the identity matrix
theorem RowExii_eq_id : RowEx i i = (1 : Matrix n n R) := by simp[RowEx]

--RowEx i j is precisely swapping the ith row of the identity matrix with the
    jth one and
--swapping the jth row of the identity row with the ith one
theorem updaterow_eq_swap [Finite n]:
updateRow (updateRow (1 : Matrix n n R) i ((1 :Matrix n n R) j)) j ((1 : Matrix
    n n R) i) =
RowEx i j := by
ext a b
by_cases ha : i = a; by_cases hb : j = b
· rw [ha,hb]
  rw[RowEx]
  rw[PEquiv.equiv_toPEquiv_toMatrix]
  rw[Equiv.swap_apply_left]
  rw[Matrix.updateRow_apply]
  rw[Matrix.updateRow_self]
  by_cases hab : a = b
  rw[hab]
  simp
  rw[if_neg hab]
· rw [ha]
  rw [RowEx]
```

```
  rw[PEquiv.equiv_toPEquiv_toMatrix]
  rw[Equiv.swap_apply_left]
  rw[Matrix.updateRow_apply]
  rw[Matrix.updateRow_self]
  by_cases haj : a = j
  rw[haj]
  rw[if_pos rfl]
  rw[if_neg haj]
· rw[RowEx]
  rw[PEquiv.equiv_toPEquiv_toMatrix]
  rw[Matrix.updateRow_apply]
  rw[Matrix.updateRow_apply]
  rw[Equiv.swap_apply_def]
  by_cases haj : a = j
  rw[if_pos]
  rw[if_neg]
  rw[if_pos]
  exact haj
  exact ne_comm.mp ha
  exact haj
  rw[if_neg haj]
  rw[if_neg]
  rw[if_neg]
  rw[if_neg haj]
  exact ne_comm.mp ha
  exact ne_comm.mp ha


-- It is commutative

theorem RowEx_comm (i j : m) :
 RowEx i j = (RowEx j i : Matrix m m R)  := by
 simp[RowEx]
 rw[Equiv.swap_comm]


--Multiplying with a matrix M with RowEx i j on the left exchanges the ith row
    and the jth row of M with each other
theorem RowExmul_eq_swap (i j: n)(M : Matrix n n R) : (RowEx i j : Matrix n n R)
    * M =
updateRow (updateRow (M) i (M j)) j (M i) := by
ext a b
by_cases ha : i = a; by_cases hb : j = b
· simp[ha,hb]
  rw[Matrix.updateRow_apply]
  rw[Matrix.updateRow_apply]
  by_cases hab : a = b;
  · rw[if_pos hab]
    rw[RowEx]
    rw[PEquiv.toPEquiv_mul_matrix]
    simp
    rw[hab]
```

```
  · rw[if_neg hab]
    rw[if_pos rfl]
    rw[RowEx]
    rw[PEquiv.toPEquiv_mul_matrix]
    simp
· rw[Matrix.updateRow_apply]
  rw[Matrix.updateRow_apply]
  rw[ha]
  by_cases haj : a = j;
  · rw[if_pos haj]
    rw[RowEx]
    rw[PEquiv.toPEquiv_mul_matrix]
    simp[haj]
  · rw[if_neg haj]
    rw[if_pos rfl]
    rw[RowEx]
    rw[PEquiv.toPEquiv_mul_matrix]
    simp
· rw[Matrix.updateRow_apply]
  rw[Matrix.updateRow_apply]
  by_cases haj : a = j;
  · rw[if_pos haj]
    rw[RowEx]
    rw[PEquiv.toPEquiv_mul_matrix]
    simp[haj]
  · rw[if_neg haj]
    rw[if_neg]
    rw[RowEx]
    rw[PEquiv.toPEquiv_mul_matrix]
    simp
    rw[Equiv.swap_apply_def]
    rw[if_neg]
    rw[if_neg haj]
    exact ne_comm.mp ha
    exact ne_comm.mp ha

theorem RowExid (i :m) (M : Matrix m m R):
(RowEx i i : Matrix m m R) * M = M := by
simp[RowExmul_eq_swap]


--RowEx i j and RowEx j i are inverses of each other
theorem RowExij_mul_Rowexji_eq_id [Finite n](i j : n): RowEx j i * RowEx i j =
    (1 : Matrix n n R) := by
rw[RowExmul_eq_swap]
rw[←updaterow_eq_swap]
rw[Matrix.updateRow_self]
ext a b
by_cases hai : i = a ; by_cases hbj : j = b
· rw[hai,hbj]
  rw[Matrix.updateRow_apply]
  rw[if_pos rfl]
```

```
· rw[hai]
  rw[Matrix.updateRow_self]
· simp[Matrix.updateRow_apply]
  rw[if_neg (ne_comm.mp hai)]
  by_cases haj : a = j;
  · rw[if_pos haj]
    rw[if_neg]
    rw[haj]
    exact Ne.trans_eq hai haj
  · rw[if_neg haj]
    rw[if_neg haj]
    rw[if_neg (ne_comm.mp hai)]


--RowEx i j is the inverse of itself
theorem RowExii_mulself_id : RowEx i j * RowEx i j = (1 : Matrix n n R) := by
rw[RowExmul_eq_swap]
rw[←updaterow_eq_swap]
rw[Matrix.updateRow_self]
ext a b
by_cases hai : i = a ; by_cases hbj : j = b
· rw[hai,hbj]
  simp[Matrix.updateRow_apply]
  intros hab hnab
  rw[hab]
  simp
· rw[hai]
  simp[Matrix.updateRow_apply]
  intros haj hnaj
  rw[← haj]
· simp[Matrix.updateRow_apply]
  by_cases haj : a = j;
  · rw[if_pos haj]
    rw[if_neg]
    rw[haj]
    exact Ne.trans_eq hai haj
  · rw[if_neg haj]
    rw[if_neg]
    rw[if_neg haj]
    rw[if_neg]
    exact ne_comm.mp hai
    exact ne_comm.mp hai



--on multiplying by RowEx i j , the jth row becomes the ith row
theorem RowExmul_applyi_eq (M : Matrix n n R) (b : n) : (RowEx i j * M:) j b = M
    i b := by
rw[RowExmul_eq_swap]
simp[updateRow_apply]

--on multiplying by RowEx i j , the ith row becomes the jth row
```

```
theorem RowExmul_applyj_eq (M : Matrix n n R) (b : n) : (RowEx i j * M:) i b = M
    j b := by
rw[RowExmul_eq_swap]
simp[updateRow_apply]
intro h
rw[h]

--on multiplying by RowEx i j , if l ≠ j and l ≠ i then the lth row remains
    unchanged
theorem RowExmul_apply_ne (b : n) (hi : i ≠ l) (hj : j ≠ l) (M : Matrix n n R):
    M l b = (RowEx i j * M:) l b :=by
rw[RowExmul_eq_swap]
simp[updateRow_apply]
rw[if_neg]
rw[if_neg]
exact id (Ne.symm hi)
exact id (Ne.symm hj)

--The determinant of RowEx i j when i ≠ j is -1
theorem RowEx_ne_det (i j : n)(h : i ≠ j): det (RowEx i j) = (-1 : R) := by
rw[RowEx]
rw[Matrix.det_permutation]
rw[Equiv.Perm.sign_swap]
simp
exact h



namespace struct

open Sum Fin TransvectionStruct Pivot Matrix
variable (R n)
--variable (M : Matrix (Sum (Fin r) Unit) (Sum (Fin r) Unit) k)

theorem rowExInl (M: Matrix (Fin r) (Fin r) k) (i j :Fin r) :
fromBlocks ((RowEx i j)*M) 0 0 (1: Matrix Unit Unit k) = (RowEx (inl i) (inl
    j))* (fromBlocks M 0 0 (1: Matrix Unit Unit k)) := by
ext a b
cases' a with a a <;> cases' b with b b
simp[RowExmul_eq_swap]
simp[Matrix.updateRow_apply]
simp[updateRow]
simp[RowExmul_eq_swap]
simp[Matrix.updateRow_apply]
simp[updateRow]
simp[RowExmul_eq_swap]
simp[Matrix.updateRow_apply]
simp[RowExmul_eq_swap]

theorem RowEx_InleqBlocks (i j : Fin r): fromBlocks (RowEx i j ) 0 0 (1: Matrix
    Unit Unit k) =
```

```
(RowEx (inl i) (inl j)) := by
suffices fromBlocks ((RowEx i j) * (1 : Matrix (Fin r) (Fin r) k)) 0 0 (1:
    Matrix Unit Unit k) =(RowEx (inl i) (inl j)) * (1 : Matrix (Fin r ⊕ Unit)
    (Fin r ⊕ Unit) k) by simpa [Matrix.mul_one]
rw[rowExInl]
rw[Matrix.mul_one]
simp

structure elimStruct where
(L : List (TransvectionStruct n R))
(i j: n)

namespace elimStruct

variable {n R}
def toElim (e : elimStruct n R) : Matrix n n R :=
((e.L).map toMatrix).prod * (RowEx e.i e.j)



def elimSum_Inl (e : elimStruct n R ) : (elimStruct (n ⊕ p) R ) where
L := ((e.L).map (sumInl p))
i := inl e.i
j := inl e.j


theorem toMat_sumInl (e : elimStruct (Fin r) k) : (e.elimSum_Inl).toElim =
    fromBlocks e.toElim 0 0 (1 : Matrix Unit Unit k) := by
simp[toElim,elimSum_Inl]
rw[←RowEx_InleqBlocks ]
rw[sumInl_toMatrix_prod_mul]

theorem toBlock_inl (M : Matrix (Fin r) (Fin r) k) (L : List (elimStruct (Fin r)
    k)) (N : Matrix Unit Unit k) (O : Matrix Unit (Fin r) k):
(L.map (toElim ∘ elimSum_Inl)).prod * fromBlocks M (0: Matrix (Fin r) Unit k) O
    N =
fromBlocks ((L.map toElim).prod * M) (0: Matrix (Fin r) Unit k) O N := by
 induction' L with t L IH
 · simp
 · simp[Matrix.mul_assoc, IH, toMat_sumInl, fromBlocks_multiply]


--variable {p}
def elimreindex (e : n ≃ p) (t : elimStruct n R) : elimStruct p R where
  L := (t.L).map (reindexEquiv e)
  i := e t.i
  j := e t.j


variable [Fintype n] [Fintype p]
```

```
theorem toElim_reindexEquiv (e : n ≃ p) (t : elimStruct n R) :
    (t.elimreindex e).toElim = (Matrix.reindexAlgEquiv R e) t.toElim := by
    rcases t with ⟨L, t_i,t_j⟩
    --ext a b
    simp only[elimreindex, transvection, mul_boole, Algebra.id.smul_eq_mul,
    toElim, toMatrix_mk, DMatrix.add_apply, Pi.smul_apply]
    have h: List.prod (List.map toMatrix (List.map (reindexEquiv e) L)) =
    List.prod (List.map (toMatrix ∘ reindexEquiv e) L) := by simp
    rw[h]
    rw[toMatrix_reindexEquiv_prod]
    rw[reindexAlgEquiv_mul]
    rw[← toMatrix_reindexEquiv_prod]
    simp only[reindexAlgEquiv_apply,reindex_apply]
    suffices RowEx (e t_i) (e t_j) = submatrix (RowEx t_i t_j) ↑e.symm ↑e.symm
    by rw[this]
    ext a b
    rw[submatrix_apply]
    by_cases ha : t_i = t_j
    have hij : e t_i = e t_j := by simp[Equiv.congr_arg ha]
    rw[hij,ha]
    simp[RowExii_eq_id,Matrix.one_apply]
    have e': p ≃ n := by apply Equiv.symm e
    have hnij : ¬ e t_i = e t_j := by
     simp[Equiv.congr_arg];exact ha
    by_cases hia : e t_i = a <;> by_cases hb : e t_j = b <;>
      simp[hia,hb,ha]
    simp[ ((e.apply_eq_iff_eq_symm_apply).mp) hia]
    simp[ ((e.apply_eq_iff_eq_symm_apply).mp) hb]
    simp[← updaterow_eq_swap,updateRow_apply]
    rw[if_neg]
    rw[if_neg]
    simp[hnij,←hia,← hb]
    exact ha
    simp[hnij,←hia,← hb]
    exact ha
    simp[ ((e.apply_eq_iff_eq_symm_apply).mp) hia]
    simp[← updaterow_eq_swap,updateRow_apply]
    rw[if_neg]
    rw[if_neg]
    rw[Matrix.one_apply_ne hb]
    rw[Matrix.one_apply_ne]
    simp[←(e.apply_eq_iff_eq_symm_apply),hb]
    simp[ha,←hia,hb,← (e.apply_eq_iff_eq_symm_apply)]
    simp[ha,←hia,hb,← (e.apply_eq_iff_eq_symm_apply)]
    simp[ ((e.apply_eq_iff_eq_symm_apply).mp) hb]
    simp[← hb]
    simp[← updaterow_eq_swap,updateRow_apply]
    by_cases haj: e t_j = a
    rw[if_pos]
    rw[if_pos]
    rw[Matrix.one_apply_ne hnij]
    rw[Matrix.one_apply_ne ha]
```

```
simp[ ((e.apply_eq_iff_eq_symm_apply).mp) haj]
rw[← haj]
rw[if_neg]
rw[if_neg]
rw[if_neg]
rw[if_neg]
rw[Matrix.one_apply_ne]
rw[Matrix.one_apply_ne]
apply ne_comm.mp
simp[← (e.apply_eq_iff_eq_symm_apply),haj]
simp[haj]
apply ne_comm.mp
push_neg at haj
exact haj
apply ne_comm.mp
simp[← (e.apply_eq_iff_eq_symm_apply),hia]
apply ne_comm.mp
simp[← (e.apply_eq_iff_eq_symm_apply),haj]
apply ne_comm.mp
push_neg  at hia
exact hia
apply ne_comm.mp
push_neg  at haj
exact haj
simp[← updaterow_eq_swap,updateRow_apply]
by_cases haj: a = e t_j
rw[if_pos]
rw[if_pos]
by_cases hbi : e t_i = b
simp[Matrix.one_apply]
rw[if_pos hbi]
rw[if_pos]
simp[ ((e.apply_eq_iff_eq_symm_apply).mp) hbi]
rw[Matrix.one_apply_ne]
rw[Matrix.one_apply_ne]
simp[← (e.apply_eq_iff_eq_symm_apply),hbi]
push_neg  at hbi
exact hbi
simp[ ((e.apply_eq_iff_eq_symm_apply).mp) (Eq.symm haj)]
exact haj
rw[if_neg haj]
rw[if_neg]
rw[if_neg]
rw[if_neg]
by_cases hab: a = b
simp[Matrix.one_apply]
simp[Matrix.one_apply]
apply ne_comm.mp
simp[← (e.apply_eq_iff_eq_symm_apply),hia]
apply ne_comm.mp
simp[← (e.apply_eq_iff_eq_symm_apply),haj]
push_neg at haj
```

```
    apply ne_comm.mp
    exact haj
    apply ne_comm.mp
    push_neg at hia
    exact hia


theorem toListElim_reindexEquiv (e : n ≃ p) (L : List (elimStruct n R)):
(L.map (toElim ∘ elimreindex e)).prod = reindexAlgEquiv R e (L.map toElim).prod
    :=by
induction' L with t L IH
· simp
· simp[IH]
  simp only [toElim_reindexEquiv]
  simp[List.prod_cons]


theorem reindex_exists (M : Matrix (Fin (Nat.succ r)) (Fin (Nat.succ r)) k ) (e
    :(Fin (Nat.succ r)) ≃  (Fin r ⊕ Unit)) (H : ∃ ( E :List (elimStruct (Fin r ⊕
     Unit) k))
 , ((E.map toElim).prod * (Matrix.reindexAlgEquiv k e M)).BlockTriangular
    OrderDual.toDual) :
 ∃ ( E :List (elimStruct (Fin (Nat.succ r)) k)), ((E.map toElim).prod *
    M).BlockTriangular OrderDual.toDual := by
 rcases H with ⟨E,H⟩
 refine' ⟨E.map (elimreindex e.symm),_⟩
 have : M = reindexAlgEquiv k e.symm (reindexAlgEquiv k e M) := by
    simp only [Equiv.symm_symm, submatrix_submatrix, reindex_apply,
    submatrix_id_id,
      Equiv.symm_comp_self, reindexAlgEquiv_apply]
 rw[this]
 simp only[List.map_map,toListElim_reindexEquiv,reindexAlgEquiv_apply]
 simp only [← reindexAlgEquiv_apply,← reindexAlgEquiv_mul]
 rw[reindexAlgEquiv_apply,reindex_apply,Equiv.symm_symm]
 have h1 :  BlockTriangular (submatrix (List.prod (List.map toElim E) *
    (reindexAlgEquiv k e) M) e e) (↑OrderDual.toDual ∘ e) :=by
  apply BlockTriangular.submatrix
  exact H
 simp[BlockTriangular]
 simp[BlockTriangular] at h1
 intro i j hij
 have h3 : ∃ c, c + i = j := by
  use j - i
  exact sub_add_cancel j i
 cases h3 with
 |intro c h3 =>
 have h2 : e (i) < e (j) := by
  simp[h3]
 apply h1 h2
--list of transvections where c is zero
```

```
def listid(k:ℕ) : List (Matrix (Sum (Fin k) Unit) (Sum (Fin k) Unit) k) :=
  List.ofFn fun i : Fin k =>
    transvection (inl i) (inr Unit.unit) (0:k)

--Product of listid is an identity matrix
theorem listid_prod_eq_id(r : ℕ) : (listid r).prod = (1 : (Matrix (Sum (Fin r)
    Unit) (Sum (Fin r) Unit) k) ) := by
simp[listid]



--For every r+1 by r+1 matrix M ,there is a list of transvections and a rowEx
    matrix such that multiplying on the left with the RowEx
--and then the list of transvections will make M_{i,1} = 0 for every 1 ≤ i < r+1
theorem transvec_RowEx_mul_lastcol (M : Matrix (Sum (Fin r) Unit) (Sum (Fin r)
    Unit) k) :
 ∃ i :Fin r ⊕ Unit, ∃ L : List (TransvectionStruct (Sum (Fin r) Unit) k), (∀ j
    : Fin r,
 ((L.map toMatrix).prod *(((RowEx i (inr unit) : Matrix (Sum (Fin r) Unit) (Sum
    (Fin r) Unit) k)) * M)) (inl j) (inr unit) = 0) := by
 --Creating two cases, when M_{1,1} is zero and non-zero
  by_cases M (inr unit) (inr unit) ≠ 0
  --First Case
  --Begin by creating the i and L that is required and inserting it in the goal
  ·let a : Fin r ⊕ Unit := inr unit
   exists a
   let L : List (TransvectionStruct (Sum (Fin r) Unit) k) :=
    List.ofFn fun i : Fin r =>
      ⟨inl i, inr unit, by simp, - (((RowEx a (inr unit) : Matrix (Sum (Fin r)
    Unit) (Sum (Fin r) Unit) k)) * M) (inl i) (inr unit) /
       (((RowEx a (inr unit) : Matrix (Sum (Fin r) Unit) (Sum (Fin r) Unit) k)) *
     M) (inr unit) (inr unit)⟩
   refine' ⟨ L,_⟩
   intro j
   --simplifying goal using listTransvecCol_mul_last_col and RowExid
   have A : L.map toMatrix = listTransvecCol (((RowEx a (inr unit) : Matrix (Sum
    (Fin r) Unit) (Sum (Fin r) Unit) k)) * M)
   := by simp [listTransvecCol, (· ∘ ·)]
   rw[A]
   simp[RowExid]
   rw[listTransvecCol_mul_last_col]
   exact h
   --Second Case
  · push_neg at h
  --Within the Second Case considering two cases when atleast one entry in last
    column is non-zero and when all entries are zero
    by_cases (∃ i : Fin r, M (inl i) (inr unit) ≠ 0)
    --2.1 Case
    · cases h with
      |intro i hi =>
      --if there is atleast one non-zero element in last column, you can make
    the M_{1,1} non-zero using RowEx
```

```
    · have hn : (((RowEx (inl i) (inr unit) : Matrix (Sum (Fin r) Unit) (Sum
  (Fin r) Unit) k) * M) (inr unit) (inr unit) ≠ 0) := by
      rw[RowExmul_eq_swap]
      rw[Matrix.updateRow_self]
      exact hi
      --Repeating a proof similar to Case 1 since M_{1,1} is non-zero
     let a : Fin r ⊕ Unit := inl i
     exists a
     let L : List (TransvectionStruct (Sum (Fin r) Unit) k) :=
     List.ofFn fun i : Fin r =>
     ⟨inl i, inr unit, by simp, - (((RowEx a (inr unit) : Matrix (Sum (Fin r)
  Unit) (Sum (Fin r) Unit) k)) * M) (inl i) (inr unit) /
     (((RowEx a (inr unit) : Matrix (Sum (Fin r) Unit) (Sum (Fin r) Unit) k))
  * M) (inr unit) (inr unit)⟩
     refine' ⟨ L,_⟩
     intro j
     have A : L.map toMatrix = listTransvecCol (((RowEx a (inr unit) : Matrix
  (Sum (Fin r) Unit) (Sum (Fin r) Unit) k)) * M)
     := by simp [listTransvecCol, (· ∘ ·)]
     rw[A]
     rw[listTransvecCol_mul_last_col]
     exact hn
  --2.2 Case
  ·push_neg at h
   let a : Fin r ⊕ Unit := inr unit
   exists a
   ---if all entries in the last column are zero L can be a list of identity
  matrices
   let L : List (TransvectionStruct (Sum (Fin r) Unit) k) :=
   List.ofFn fun i : Fin r =>
   ⟨inl i, inr unit, by simp, 0 ⟩
   exists L
   intro j
   have A : L.map toMatrix = listid r := by
    simp[listid, (· ∘ ·)]
   rw[A,listid_prod_eq_id]
   simp[h,RowExid]
```

```
theorem exists_elimmatrix_mul_lastcol (M : Matrix (Sum (Fin r) Unit) (Sum (Fin
    r) Unit) k)  :
∃(N : elimStruct (Fin r ⊕ Unit) k), (∀ j : Fin r , ((N.toElim) * M) (inl j)
    (inr unit) = 0) :=by
 · have TH :  ∃ i :Fin r ⊕ Unit, ∃ L : List (TransvectionStruct (Sum (Fin r)
    Unit) k), (∀ j : Fin r,
   ((L.map toMatrix).prod *(((RowEx i (inr unit) : Matrix (Sum (Fin r) Unit)
    (Sum (Fin r) Unit) k)) * M)) (inl j) (inr unit) = 0):=by
```

```
    apply transvec_RowEx_mul_lastcol M
   rcases TH with ⟨ i,L',TH⟩
   simp[toElim]
   let N': elimStruct (Fin r ⊕ Unit) k :=
   ⟨L',i,(inr unit)⟩
   exists N'
   simp[N']
   suffices  ∀ (j : Fin r),
   (List.prod (List.map toMatrix L') * ((RowEx k (inr unit) : Matrix (Sum (Fin
    r) Unit) (Sum (Fin r) Unit) k) * M)) (inl j) (inr unit)
    = 0 by
    simp[Matrix.mul_assoc]
    exact TH
   exact TH

end elimStruct

open elimStruct



theorem exists_Listelimmatrix_eq_lowertriangular(IH : ∀ (M : Matrix (Fin r) (Fin
    r) k), ∃ ( E :List (elimStruct (Fin r) k))
 , ((E.map toElim).prod * M).BlockTriangular OrderDual.toDual) (M :Matrix (Sum
    (Fin r) Unit) (Sum (Fin r) Unit) k) :
  ∃ (E₁ : List (elimStruct (Fin r ⊕ Unit) k) ),
  ((E₁.map toElim).prod * M).BlockTriangular OrderDual.toDual := by
  have HM : ∃ N : elimStruct (Fin r ⊕ Unit) k, ∀ (j : Fin r), (toElim N * M)
    (inl j) (inr Unit.unit) = 0 := by
   apply exists_elimmatrix_mul_lastcol M
  cases HM with
  |intro N HM =>
  let M' := N.toElim*M
  let M'' := toBlocks₁₁ M'
  rcases IH M'' with ⟨L, h₀⟩
  set Mₐ := toBlocks₂₁ M'
  set c := toBlocks₂₂ M'
  refine'⟨L.map (elimSum_Inl) ++ [N],_⟩
  suffices ((L.map (toElim ∘ elimSum_Inl)).prod * M').BlockTriangular
    OrderDual.toDual by simpa[Matrix.mul_assoc]
  · have H : M' = fromBlocks (M'') 0 Mₐ c := by
     simp
     have X : toBlocks₁₂ (M') = 0 := by
      ext a b
      simp[toBlocks₁₂]
      rw[HM a]
     rw[←X]
     simp[Matrix.fromBlocks_toBlocks]
    rw[H]
    rw[toBlock_inl]
    simpa[BlockTriangular]
```

```
theorem gauss_elim (M' : Matrix (Fin r) (Fin r) k) : ∃ ( E :List (elimStruct
    (Fin r) k))
 , ((E.map toElim).prod * M').BlockTriangular OrderDual.toDual := by
 induction' r with n IH
 refine' ⟨List.nil,_⟩
 simp[BlockTriangular]
 have h1 : Fintype.card (Fin (Nat.succ n))  = Fintype.card (Fin n ⊕ Unit) := by
  rw [@Fintype.card_sum (Fin n) Unit _ _]
  simp
 rw[Fintype.card_eq] at h1
 apply Nonempty.some at h1
 apply reindex_exists M' h1
 apply exists_Listelimmatrix_eq_lowertriangular IH
```