# Rank-Nullity Theorem in Linear Algebra 

By Jose Divasón and Jesús Aransay*

March 11, 2024


#### Abstract

In this contribution, we present some formalizations based on the HOL-Multivariate-Analysis session of Isabelle. Firstly, a generalization of several theorems of such library are presented. Secondly, some definitions and proofs involving Linear Algebra and the four fundamental subspaces of a matrix are shown. Finally, we present a proof of the result known in Linear Algebra as the "Rank-Nullity Theorem", which states that, given any linear map $f$ from a finite dimensional vector space $V$ to a vector space $W$, then the dimension of $V$ is equal to the dimension of the kernel of $f$ (which is a subspace of $V$ ) and the dimension of the range of $f$ (which is a subspace of $W$ ). The proof presented here is based on the one given in [1]. As a corollary of the previous theorem, and taking advantage of the relationship between linear maps and matrices, we prove that, for every matrix $A$ (which has associated a linear map between finite dimensional vector spaces), the sum of its null space and its column space (which is equal to the range of the linear map) is equal to the number of columns of $A$.


## Contents

## 1 Dual Order

1.1 Interpretation of dual wellorder based on wellorder ..... 2
1.2 Properties of the Greatest operator ..... 3
2 Class for modular arithmetic ..... 3
2.1 Definition and properties ..... 3
2.2 Conversion between a modular class and the subset of natural numbers associated. ..... 4
2.3 Instantiations ..... 13

[^0]3 Miscellaneous ..... 15
3.1 Definitions of number of rows and columns of a matrix ..... 15
3.2 Basic properties about matrices ..... 15
3.3 Theorems obtained from the AFP ..... 16
3.4 Basic properties involving span, linearity and dimensions ..... 18
3.5 Basic properties about matrix multiplication ..... 19
3.6 Properties about invertibility ..... 20
3.7 Properties about the dimension of vectors ..... 21
3.8 Instantiations and interpretations ..... 22
4 Fundamental Subspaces ..... 23
4.1 The fundamental subspaces of a matrix ..... 23
4.1.1 Definitions ..... 23
4.1.2 Relationships among them ..... 23
4.2 Proving that they are subspaces ..... 23
4.3 More useful properties and equivalences ..... 24
5 Rank Nullity Theorem of Linear Algebra ..... 27
5.1 Previous results ..... 27
5.2 The proof ..... 29
5.3 The rank nullity theorem for matrices ..... 32

## 1 Dual Order

```
theory Dual-Order
    imports Main
begin
```

```
1.1 Interpretation of dual wellorder based on wellorder
lemma wf-wellorderI2:
    assumes wf:wf {(x::'a::ord, y). y<x}
    assumes lin: class.linorder ( }\lambda(x:\mp@subsup{:}{}{\prime}a) y::'a. y \leqx) (\lambda(x::'a) y::'a. y<x
    shows class.wellorder ( }\lambda(x:\mp@subsup{:}{}{\prime}a) y::'a. y \leqx) (\lambda(x::'a) y::'a. y<x
    using lin unfolding class.wellorder-def apply (rule conjI)
    apply (rule class.wellorder-axioms.intro) by (blast intro: wf-induct-rule [OF wf])
interpretation dual-wellorder:wellorder ( }\geq\mathrm{ )::('a::{linorder, finite } =>' }a=>>bool
(>)
proof (rule wf-wellorderI2)
    show wf {(x :: 'a,y). y<x}
        by(auto simp add: trancl-def intro!: finite-acyclic-wf acyclicI)
    show class.linorder (\lambda(x::'a) y::'a. y \leqx) (\lambda(x::'a) y::'a. y<x)
        unfolding class.linorder-def unfolding class.linorder-axioms-def unfolding
class.order-def
    unfolding class.preorder-def unfolding class.order-axioms-def by auto
qed
```


### 1.2 Properties of the Greatest operator

```
lemma dual-wellorder-Least-eq-Greatest[simp]: dual-wellorder.Least = Greatest
    by (auto simp add: Greatest-def dual-wellorder.Least-def)
lemmas GreatestI = dual-wellorder.LeastI[unfolded dual-wellorder-Least-eq-Greatest]
lemmas GreatestI2-ex = dual-wellorder.LeastI2-ex[unfolded dual-wellorder-Least-eq-Greatest]
lemmas GreatestI2-wellorder = dual-wellorder.LeastI2-wellorder[unfolded dual-wellorder-Least-eq-Greatest]
lemmas GreatestI-ex = dual-wellorder.LeastI-ex[unfolded dual-wellorder-Least-eq-Greatest]
lemmas not-greater-Greatest = dual-wellorder.not-less-Least[unfolded dual-wellorder-Least-eq-Greatest]
lemmas GreatestI2 = dual-wellorder.LeastI2[unfolded dual-wellorder-Least-eq-Greatest]
lemmas Greatest-ge = dual-wellorder.Least-le[unfolded dual-wellorder-Least-eq-Greatest]
```

end

## 2 Class for modular arithmetic

```
theory Mod-Type
imports
    HOL-Library.Numeral-Type
    HOL-Analysis.Cartesian-Euclidean-Space
    Dual-Order
begin
```


### 2.1 Definition and properties

Class for modular arithmetic. It is inspired by the locale mod_type.

```
class mod-type = times + wellorder + neg-numeral +
fixes Rep :: 'a => int
    and Abs:: int => 'a
    assumes type: type-definition Rep Abs {0..<int CARD ('a)}
    and size1:1 < int CARD ('a)
    and zero-def: 0 = Abs 0
    and one-def: 1 = Abs 1
    and add-def: }x+y=Abs((Rep x + Rep y) mod (int CARD ('a)))
    and mult-def:x*y=Abs ((Rep x*Rep y) mod (int CARD ('a)))
    and diff-def: x - y=Abs ((Rep x - Rep y) mod (int CARD ('a)))
    and minus-def: - x = Abs ((- Rep x) mod (int CARD ('a)))
    and strict-mono-Rep: strict-mono Rep
begin
lemma size0: 0< int CARD ('a)
    using size1 by simp
lemmas definitions =
    zero-def one-def add-def mult-def minus-def diff-def
lemma Rep-less-n: Rep x < int CARD ('a)
    by (rule type-definition.Rep [OF type, simplified, THEN conjunct2])
```

```
lemma Rep-le-n: Rep \(x \leq \operatorname{int} \operatorname{CARD}\) ('a)
    by (rule Rep-less-n [THEN order-less-imp-le])
lemma Rep-inject-sym: \(x=y \longleftrightarrow\) Rep \(x=\operatorname{Rep} y\)
    by (rule type-definition.Rep-inject [OF type, symmetric])
lemma Rep-inverse: Abs \((\operatorname{Rep} x)=x\)
    by (rule type-definition.Rep-inverse [OF type])
lemma Abs-inverse: \(m \in\left\{0 . .<\operatorname{int} C A R D\left({ }^{\prime} a\right)\right\} \Longrightarrow \operatorname{Rep}(A b s m)=m\)
    by (rule type-definition.Abs-inverse [OF type])
lemma Rep-Abs-mod: Rep (Abs ( \(m\) mod int CARD ('a))) \(=m \bmod \operatorname{int} C A R D ~(' a)\)
    using size0 by (auto simp add: Abs-inverse)
lemma Rep-Abs-0: Rep (Abs 0) \(=0\)
    apply (rule Abs-inverse [of 0])
    using size0 by simp
lemma Rep-0: Rep \(0=0\)
    by (simp add: zero-def Rep-Abs-0)
lemma Rep-Abs-1: Rep (Abs 1) = 1
    by (simp add: Abs-inverse size1)
lemma Rep-1: Rep \(1=1\)
    by (simp add: one-def Rep-Abs-1)
lemma Rep-mod: Rep \(x\) mod int \(C A R D(' a)=\operatorname{Rep} x\)
    apply (rule-tac \(x=x\) in type-definition.Abs-cases [OF type])
    apply (simp add: type-definition.Abs-inverse [OF type])
done
lemmas Rep-simps \(=\)
    Rep-inject-sym Rep-inverse Rep-Abs-mod Rep-mod Rep-Abs-0 Rep-Abs-1
```


### 2.2 Conversion between a modular class and the subset of natural numbers associated.

Definitions to make transformations among elements of a modular class and naturals

```
definition to-nat :: 'a => nat
    where \(t o-n a t=n a t \circ R e p\)
definition \(A b s^{\prime}::\) int \(=>~ ' a\)
    where \(A b s^{\prime} x=A b s(x\) mod int CARD ('a))
```

definition from-nat :: nat $\Rightarrow{ }^{\prime} a$
where from-nat $=\left(A b s^{\prime} \circ\right.$ int $)$
lemma bij-Rep: bij-betw (Rep) (UNIV::'a set) \{0..<int CARD('a)\}
proof (unfold bij-betw-def, rule conjI)
show inj Rep by (metis strict-mono-imp-inj-on strict-mono-Rep)
show range Rep $=\left\{0 . .<\right.$ int $\left.\operatorname{CARD}\left({ }^{\prime} a\right)\right\}$ using Typedef.type-definition.Rep-range $[O F$ type] .
qed
lemma mono-Rep: mono Rep by (metis strict-mono-Rep strict-mono-mono)
lemma Rep-ge-0: $0 \leq \operatorname{Rep} x$ using bij-Rep unfolding bij-betw-def by auto
lemma bij-Abs: bij-betw (Abs) \{0..<int CARD('a)\} (UNIV::'a set)
proof (unfold bij-betw-def, rule conjI)
show inj-on Abs $\{0 . .<$ int $C A R D(' a)\}$ by (metis inj-on-inverseI type type-definition.Abs-inverse)
show Abs ' $\left\{0 . .<\right.$ int $\left.C A R D\left({ }^{\prime} a\right)\right\}=(U N I V:: ' a$ set $)$ by (metis type type-definition.univ)
qed
corollary bij-Abs': bij-betw $(A b s ')\{0 . .<$ int $C A R D(' a)\}(U N I V:: ' a$ set)
proof (unfold bij-betw-def, rule conjI)
show inj-on $A b s^{\prime}\left\{0 . .<\right.$ int $\left.\operatorname{CARD}\left({ }^{\prime} a\right)\right\}$
unfolding inj-on-def Abs'-def
by (auto, metis Rep-Abs-mod mod-pos-pos-trivial)
show $A b s^{\prime}$ ' $\left\{0 . .<\right.$ int $\left.C A R D\left({ }^{\prime} a\right)\right\}=\left(U N I V::^{\prime} a\right.$ set $)$
proof (unfold image-def Abs'-def, auto)
fix $x$ show $\exists x a \in\{0 . .<$ int $C A R D(' a)\} . x=A b s x a$
by (rule bexI[of-Rep x], auto simp add: Rep-less-n[of x] Rep-ge-0[of x], metis Rep-inverse)
qed
qed
lemma bij-from-nat: bij-betw (from-nat) $\left\{0 . .<C A R D\left({ }^{\prime} a\right)\right\}$ (UNIV ::'a set)
proof (unfold bij-betw-def, rule conjI)
have set-eq: $\left\{0::\right.$ int.. $<$ int $\left.C A R D\left({ }^{\prime} a\right)\right\}=$ int $^{\prime}\left\{0 . .<C A R D\left({ }^{\prime} a\right)\right\}$ apply (auto)
proof -
fix $x::$ int assume $x 1:(0::$ int $) \leq x$ and $x 2: x<$ int $C A R D\left({ }^{\prime} a\right)$ show $x \in$ int
' $\left\{0::\right.$ nat.. $\left.<\operatorname{CARD}\left({ }^{\prime} a\right)\right\}$
proof (unfold image-def, auto, rule bexI[of - nat x])
show $x=$ int (nat $x$ ) using $x 1$ by auto
show nat $x \in\left\{0::\right.$ nat.. $\left.<C A R D\left({ }^{\prime} a\right)\right\}$ using $x 1$ x2 by auto
qed
qed
show inj-on (from-nat::nat $\left.\Rightarrow^{\prime} a\right)\left\{0:: n a t . .<C A R D\left({ }^{\prime} a\right)\right\}$
proof (unfold from-nat-def, rule comp-inj-on)
show inj-on int $\{0:: n a t . .<C A R D(' a)\}$ by (metis inj-of-nat subset-inj-on top-greatest)
show inj-on (Abs'::int=>'a) (int ' $\{0:: n a t . .<C A R D(' a)\})$
using bij-Abs unfolding bij-betw-def set-eq
by (metis (opaque-lifting, no-types) Abs'-def Abs-inverse Rep-inverse Rep-mod inj-on-def set-eq)
qed
show (from-nat::nat $\left.=>^{\prime} a\right)^{\prime}\left\{0:: n a t . .<C A R D\left({ }^{\prime} a\right)\right\}=U N I V$
unfolding from-nat-def using bij-Abs ${ }^{\prime}$
unfolding bij-betw-def set-eq o-def by blast
qed
lemma to-nat-is-inv: the-inv-into $\left\{0 . .<C A R D\left({ }^{\prime} a\right)\right\}\left(\right.$ from-nat::nat $\left.=>^{\prime} a\right)=($ to-nat::'a=>nat)
proof (unfold the-inv-into-def fun-eq-iff from-nat-def to-nat-def o-def, clarify)
fix $x::^{\prime} a$ show (THE $y::$ nat. $y \in\left\{0::\right.$ nat.. $\left.<\operatorname{CARD}\left({ }^{\prime} a\right)\right\} \wedge A b s^{\prime}($ int $\left.y)=x\right)=$ nat (Rep $x$ )
proof (rule the-equality, auto)
show $A b s^{\prime}($ Rep $x)=x$ by (metis Abs'-def Rep-inverse Rep-mod)
show nat (Rep $x)<\operatorname{CARD}\left(^{\prime} a\right)$ by (metis (full-types) Rep-less-n nat-int size0 zless-nat-conj)
assume $x: \neg(0::$ int $) \leq \operatorname{Rep} x$ show $(0:: n a t)<C A R D\left({ }^{\prime} a\right)$ and $A b s^{\prime}(0::$ int $)$
$=x$
using Rep-ge-0 $x$ by auto
next
fix $y$ ::nat assume $y: y<C A R D\left({ }^{\prime} a\right)$
have $\left(\operatorname{Rep}\left(A b s^{\prime}(\right.\right.$ int $\left.\left.y)::^{\prime} a\right)\right)=\left(\operatorname{Rep}\left(\left(\operatorname{Abs}\left(\right.\right.\right.\right.$ int $\left.\left.\left.\left.y \bmod \operatorname{int} C A R D\left({ }^{\prime} a\right)\right)\right)::^{\prime} a\right)\right)$ un-
folding $A b s^{\prime}$-def ..
also have $\ldots=\left(\operatorname{Rep}\left(\operatorname{Abs}(\right.\right.$ int $\left.\left.y)::^{\prime} a\right)\right)$ using zmod-int $\left[\right.$ of $\left.y \operatorname{CARD}\left({ }^{\prime} a\right)\right]$
using $y$ mod-less by auto
also have $\ldots=($ int $y)$ proof (rule Abs-inverse) show int $y \in\{0::$ int.. $<$ int $\left.C A R D\left({ }^{\prime} a\right)\right\}$
using $y$ by auto qed
finally show $y=$ nat $\left(\operatorname{Rep}\left(A b s^{\prime}(\right.\right.$ int $\left.\left.y)::^{\prime} a\right)\right)$ by (metis nat-int)
qed
qed
lemma bij-to-nat: bij-betw (to-nat) (UNIV::'a set) $\{0 . .<C A R D(' a)\}$
using bij-betw-the-inv-into[OF bij-from-nat] unfolding to-nat-is-inv.
lemma finite-mod-type: finite (UNIV ::'a set)
using finite-imageD[of to-nat UNIV ::'a set] using bij-to-nat unfolding bij-betw-def
by auto
subclass (in mod-type) finite by (intro-classes, rule finite-mod-type)
lemma least-0: (LEAST n. $n \in\left(\right.$ UNIV $::^{\prime} a$ set $\left.)\right)=0$
proof (rule Least-equality, auto)
fix $y$ : : ${ }^{\prime} a$
have $\left(0::^{\prime} a\right) \leq A b s$ (Rep y mod int CARD('a)) using strict-mono-Rep unfolding strict-mono-def
by (metis (opaque-lifting, mono-tags) Rep-0 Rep-ge-0 strict-mono-Rep strict-mono-less-eq) also have $\ldots=y$ by (metis Rep-inverse Rep-mod)
finally show $\left(0::^{\prime} a\right) \leq y$.

## qed

lemma add-to-nat-def: $x+y=$ from-nat (to-nat $x+$ to-nat $y$ )
unfolding from-nat-def to-nat-def o-def using Rep-ge-0[of x] using Rep-ge-O[of $y]$
using Rep-less-n[of x] Rep-less-n[of y]
unfolding $A b s^{\prime}$-def unfolding add-def $[o f x y]$ by auto
lemma to-nat-1: to-nat $1=1$
by (simp add: to-nat-def Rep-1)
lemma add-def':
shows $x+y=A b s^{\prime}(\operatorname{Rep} x+\operatorname{Rep} y)$ unfolding $A b s^{\prime}$-def using add-def by simp
lemma $A b s^{\prime}-0$ :
shows $A b s^{\prime}\left(C A R D\left({ }^{\prime} a\right)\right)=\left(0::^{\prime} a\right)$ by (metis (opaque-lifting, mono-tags) $A b s^{\prime}$-def mod-self zero-def)
lemma Rep-plus-one-le-card:
assumes $a: a+1 \neq 0$
shows (Rep a) $+1<\operatorname{CARD}$ ('a)
proof (rule ccontr)
assume $\neg \operatorname{Rep} a+1<C A R D\left({ }^{\prime} a\right)$ hence to-nat-eq-card: Rep $a+1=C A R D\left({ }^{\prime} a\right)$ using Rep-less-n by (simp add: add1-zle-eq order-class.less-le)
have $a+1=A b s^{\prime}\left(\operatorname{Rep} a+\operatorname{Rep}\left(1::^{\prime} a\right)\right)$ using add-def' by auto
also have $\ldots=A b s^{\prime}((\operatorname{Rep} a)+1)$ using Rep-1 by simp
also have $\ldots=A b s^{\prime}\left(C A R D\left({ }^{\prime} a\right)\right)$ unfolding to-nat-eq-card ..
also have $\ldots=0$ using $A b s^{\prime}-0$ by auto
finally show False using $a$ by contradiction
qed
lemma to-nat-plus-one-less-card: $\forall a . a+1 \neq 0-->$ to-nat $a+1<C A R D\left({ }^{\prime} a\right)$ proof (clarify)
fix $a$
assume $a: a+1 \neq 0$
have Rep $a+1<\mathrm{int} C A R D\left({ }^{\prime} a\right)$ using Rep-plus-one-le-card $[O F a]$ by auto hence nat (Rep $a+1$ ) < nat (int CARD('a)) unfolding zless-nat-conj using size0 by fast
thus to-nat $a+1<C A R D\left({ }^{\prime} a\right)$ unfolding to-nat-def o-def using nat-add-distrib[OF Rep-ge-0] by simp
qed
corollary to-nat-plus-one-less-card ${ }^{\prime}$ :
assumes $a+1 \neq 0$
shows to-nat $a+1<C A R D(' a)$ using to-nat-plus-one-less-card assms by simp
lemma strict-mono-to-nat: strict-mono to-nat
using strict-mono-Rep
unfolding strict-mono-def to-nat-def using Rep-ge-0 by (metis comp-apply nat-less-eq-zless)
lemma to-nat-eq [simp]: to-nat $x=$ to-nat $y \longleftrightarrow x=y$
using injD [OF bij-betw-imp-inj-on[OF bij-to-nat]] by blast
lemma mod-type-forall-eq [simp]: $\left(\forall j::^{\prime} a .(\right.$ to-nat $\left.j)<C A R D(' a) \longrightarrow P j\right)=(\forall a$. $P a)$ proof (auto)
fix $a$ assume $a: \forall j$. (to-nat::' $a=>n a t) j<C A R D\left({ }^{\prime} a\right) \longrightarrow P j$
have (to-nat::'a=>nat) $a<C A R D\left({ }^{\prime} a\right)$ using bij-to-nat unfolding bij-betw-def by auto
thus $P$ a using $a$ by auto
qed
lemma to-nat-from-nat:
assumes $t$ :to-nat $j=k$
shows from-nat $k=j$
proof -
have from-nat $k=$ from-nat (to-nat $j$ ) unfolding $t$..
also have $\ldots=$ from-nat (the-inv-into $\left\{0 . .<\operatorname{CARD}\left({ }^{\prime} a\right)\right\}$ (from-nat) j) unfolding to-nat-is-inv ..
also have $\ldots=j$
proof (rule f-the-inv-into-f)
show inj-on from-nat $\left\{0 . .<C A R D\left({ }^{\prime} a\right)\right\}$ by (metis bij-betw-imp-inj-on bij-from-nat)
show $j \in$ from-nat ' $\left\{0 . .<C A R D\left({ }^{\prime} a\right)\right\}$ by (metis UNIV-I bij-betw-def bij-from-nat)
qed
finally show from-nat $k=j$.
qed
lemma to-nat-mono:
assumes $a b: a<b$
shows to-nat $a<$ to-nat $b$
using strict-mono-to-nat unfolding strict-mono-def using assms by fast
lemma to-nat-mono':
assumes $a b: a \leq b$
shows to-nat $a \leq$ to-nat $b$
proof (cases $a=b$ )
case True thus ?thesis by auto
next
case False
hence $a<b$ using $a b$ by simp
thus ?thesis using to-nat-mono by fastforce
qed
lemma least-mod-type:
shows $0 \leq\left(n::^{\prime} a\right)$

```
    using least-0 by (metis (full-types) Least-le UNIV-I)
lemma to-nat-from-nat-id:
    assumes x: x<CARD('a)
    shows to-nat ((from-nat x)::'a) = x
    unfolding to-nat-is-inv[symmetric] proof (rule the-inv-into-f-f)
    show inj-on (from-nat::nat=>'a) {0..<CARD('a)} using bij-from-nat unfold-
ing bij-betw-def by auto
    show }x\in{0..<CARD('a)} using x by sim
qed
lemma from-nat-to-nat-id[simp]:
    shows from-nat (to-nat x) =x by (metis to-nat-from-nat)
lemma from-nat-to-nat:
    assumes t:from-nat j = k and j: j<CARD('a)
    shows to-nat k=j by (metis jt to-nat-from-nat-id)
lemma from-nat-mono:
    assumes i-le-j:i<j and j: j<CARD('a)
    shows (from-nat i::'a) < from-nat j
proof -
have i: i<CARD('a) using i-le-j j by simp
obtain a where a: i=to-nat a
    using bij-to-nat unfolding bij-betw-def using i to-nat-from-nat-id by metis
obtain b where b: j=to-nat b
    using bij-to-nat unfolding bij-betw-def using j to-nat-from-nat-id by metis
show ?thesis by (metis a b from-nat-to-nat-id i-le-j strict-mono-less strict-mono-to-nat)
qed
lemma from-nat-mono':
    assumes i-le-j:i\leqj and j<CARD ('a)
    shows (from-nat i::'a) \leq from-nat j
proof (cases i=j)
    case True
    have (from-nat i::'a)= from-nat j using True by simp
    thus ?thesis by simp
next
    case False
    hence i<j using i-le-j by simp
    thus ?thesis by (metis assms(2) from-nat-mono less-imp-le)
qed
lemma to-nat-suc:
    assumes to-nat (x)+1<CARD ('a)
    shows to-nat (x+1::'a)=(to-nat x)+1
proof -
    have (x::'a) + 1 = from-nat (to-nat x + to-nat (1::'a)) unfolding add-to-nat-def
```

```
    hence to-nat ((x::'a) + 1) = to-nat (from-nat (to-nat x + to-nat (1::'a))::'a)
by presburger
    also have ... = to-nat (from-nat (to-nat x + 1)::'a) unfolding to-nat-1 ..
    also have ... = (to-nat x + 1) by (metis assms to-nat-from-nat-id)
    finally show ?thesis.
qed
lemma to-nat-le:
    assumes }y<\mathrm{ from-nat }
    shows to-nat y<k
proof (cases k<CARD('a))
    case True show ?thesis by (metis (full-types) True assms to-nat-from-nat-id
to-nat-mono)
next
    case False have to-nat y < CARD ('a) using bij-to-nat unfolding bij-betw-def
by auto
    thus ?thesis using False by auto
qed
lemma le-Suc:
    assumes ab: a< (b::'a)
    shows a+1\leqb
proof -
    have a + = (from-nat (to-nat (a+1))::'a) using from-nat-to-nat-id [of
a+1,symmetric].
    also have ... \leq(from-nat (to-nat (b::'a))::'a)
    proof (rule from-nat-mono')
    have to-nat a < to-nat b using ab by (metis to-nat-mono)
    hence to-nat a+1\leq to-nat b by simp
    thus to-nat b < CARD ('a) using bij-to-nat unfolding bij-betw-def by auto
            hence to-nat a+1<CARD ('a) by (metis <to-nat a + 1 土 to-nat b>
preorder-class.le-less-trans)
    thus to-nat (a+1)\leqto-nat b by (metis <to-nat a + 1 \leq to-nat b> to-nat-suc)
    qed
    also have ... = b by (metis from-nat-to-nat-id)
    finally show }a+(1::'a)\leqb 
qed
lemma le-Suc':
assumes ab: a+1\leqb
    and less-card:(to-nat a)+1<CARD ('a)
    shows a<b
proof -
    have }a=(from-nat (to-nat a)::'a) using from-nat-to-nat-id [of a,symmetric] .
    also have ... < (from-nat (to-nat b)::'a)
    proof (rule from-nat-mono)
    show to-nat b < CARD('a) using bij-to-nat unfolding bij-betw-def by auto
    have to-nat (a+1)\leq to-nat b using ab by (metis to-nat-mono')
    hence to-nat (a)+1\leq to-nat b using to-nat-suc[OF less-card] by auto
```

```
        thus to-nat a<to-nat b by simp
    qed
    finally show }a<b\mathrm{ by (metis to-nat-from-nat)
qed
lemma Suc-le:
    assumes less-card: (to-nat a) + < CARD ('a)
    shows }a<a+
proof -
    have (to-nat a)<(to-nat a)+1 by simp
    hence (to-nat a) < to-nat (a+1) by (metis less-card to-nat-suc)
    hence (from-nat (to-nat a)::'a) < from-nat (to-nat (a+1))
    by (rule from-nat-mono, metis less-card to-nat-suc)
    thus }a<a+1\mathrm{ by (metis to-nat-from-nat)
qed
lemma Suc-le':
    fixes a::'a
    assumes }a+1\not=
    shows }a<a+1\mathrm{ using Suc-le to-nat-plus-one-less-card assms by blast
lemma from-nat-not-eq:
    assumes a-eq-to-nat:a\not= to-nat b
    and a-less-card: a<CARD('a)
    shows from-nat a}\not=
proof (rule ccontr)
    assume \neg from-nat }a\not=b\mathrm{ hence from-nat }a=b\mathrm{ by simp
    hence to-nat ((from-nat a)::'a) = to-nat b by auto
    thus False by (metis a-eq-to-nat a-less-card to-nat-from-nat-id)
qed
lemma Suc-less:
    fixes i::'a
    assumes i<j
    and i+1\not=j
    shows i+1<j by (metis assms le-Suc le-neq-trans)
lemma Greatest-is-minus-1: \foralla::'a. a \leq -1
proof (clarify)
    fix }a::'
    have zero-ge-card-1:0 \leq int CARD('a) - 1 using size1 by auto
    have card-less: int CARD('a) - 1 < int CARD('a) by auto
    have not-zero: }1\mathrm{ mod int CARD('a)}\not=
    by (metis (opaque-lifting, mono-tags) Rep-Abs-1 Rep-mod zero-neq-one)
    have int-card: int (CARD('a) - 1) = int CARD('a) - 1 using of-nat-diff[of 1
CARD ('a)]
    using size1 by simp
    have a=Abs'(Rep a) by (metis (opaque-lifting, mono-tags) Rep-0 add-0-right
```

```
add-def'
        monoid-add-class.add.right-neutral)
    also have ... = Abs' (int (nat (Rep a))) by (metis Rep-ge-0 int-nat-eq)
    also have ... \leqAbs' (int (CARD('a) - 1))
    proof (rule from-nat-mono'[unfolded from-nat-def o-def, of nat (Rep a) CARD('a)
- 1])
    show nat (Rep a) \leqCARD('a) - 1 using Rep-less-n
        using int-card nat-le-iff by auto
    show CARD('a) - 1 < CARD('a) using finite-UNIV-card-ge-0 finite-mod-type
by fastforce
    qed
    also have ... = - 1
    unfolding Abs'-def unfolding minus-def zmod-zminus1-eq-if unfolding Rep-1
    apply (rule cong [of Abs], rule refl)
    unfolding if-not-P [OF not-zero]
    unfolding int-card
    unfolding mod-pos-pos-trivial[OF zero-ge-card-1 card-less]
    using mod-pos-pos-trivial[OF - size1] by presburger
    finally show }a\leq-1\mathrm{ by fastforce
qed
lemma a-eq-minus-1: }\foralla:\mp@subsup{:}{}{\prime}a.a+1=0\longrightarrowa=-
    by (metis eq-neg-iff-add-eq-0)
lemma forall-from-nat-rw:
    shows (\forallx\in{0..<CARD('a)}.P(from-nat x::'a))}=(\forallx.P(\mathrm{ from-nat }x)
proof (auto)
    fix y assume *: }\forallx\in{0..<CARD('a)}. P (from-nat x
    have from-nat y (UNIV::'a set) by auto
    from this obtain x where x1: from-nat y = (from-nat x::'a) and x2: x\in{0..<CARD('a)}
        using bij-from-nat unfolding bij-betw-def
        by (metis from-nat-to-nat-id rangeI the-inv-into-onto to-nat-is-inv)
    show P (from-nat y::'a) unfolding x1 using * x2 by simp
qed
lemma from-nat-eq-imp-eq:
    assumes f-eq: from-nat x = (from-nat xa::'a)
and x: x<CARD('a) and xa: xa<CARD('a)
    shows x=xa using assms from-nat-not-eq by metis
lemma to-nat-less-card
    fixes j::'a
    shows to-nat j < CARD ('a)
    using bij-to-nat unfolding bij-betw-def by auto
lemma from-nat-0: from-nat 0 = 0
    unfolding from-nat-def o-def of-nat-0 Abs'-def mod-0 zero-def ..
lemma to-nat-0: to-nat 0 = 0 unfolding to-nat-def o-def Rep-0 nat-0 ..
```

```
lemma to-nat-eq-0:(to-nat x = 0) =(x=0)
    by (auto simp add: to-nat-0 from-nat-0 dest: to-nat-from-nat)
lemma suc-not-zero:
    assumes to-nat a + 1 = CARD('a)
    shows }a+1\not=
proof (rule ccontr, simp)
    assume a-plus-one-zero: }a+1=
    hence rep-eq-card: Rep a+1=CARD('a)
    using assms to-nat-0 Suc-eq-plus1 Suc-lessI Zero-not-Suc to-nat-less-card to-nat-suc
    by (metis (opaque-lifting, mono-tags))
    moreover have Rep a + < < CARD('a)
        using Abs'-0 Rep-1 Suc-eq-plus1 Suc-lessI Suc-neq-Zero add-def' assms
    rep-eq-card to-nat-0 to-nat-less-card to-nat-suc by (metis (opaque-lifting, mono-tags))
    ultimately show False by fastforce
qed
lemma from-nat-suc:
shows from-nat (j+1) = from-nat j + 1
unfolding from-nat-def o-def Abs'-def add-def' Rep-1 Rep-Abs-mod
unfolding of-nat-add apply (subst mod-add-left-eq) unfolding of-nat-1 ..
lemma to-nat-plus-1-set:
shows to-nat a + 1 \in{1..<CARD('a)+1}
using to-nat-less-card by simp
end
lemma from-nat-CARD:
    shows from-nat (CARD('a)) = (0::'a::{mod-type})
    unfolding from-nat-def o-def Abs'-def by (simp add: zero-def)
```


### 2.3 Instantiations

```
instantiation bit0 and bit1:: (finite) mod-type begin
```

```
definition (Rep::'a bit0 => int) x= Rep-bit0 x
```

definition (Rep::'a bit0 => int) x= Rep-bit0 x
definition (Abs::int => 'a bit0) x=Abs-bit0' }
definition (Abs::int => 'a bit0) x=Abs-bit0' }
definition (Rep::'a bit1 => int) x = Rep-bit1 x
definition (Rep::'a bit1 => int) x = Rep-bit1 x
definition (Abs::int => 'a bit1) x = Abs-bit1' }
definition (Abs::int => 'a bit1) x = Abs-bit1' }
instance
instance
proof
proof
show (0::'a bit0) = Abs (0::int) unfolding Abs-bit0-def Abs-bit0'-def zero-bit0-def
show (0::'a bit0) = Abs (0::int) unfolding Abs-bit0-def Abs-bit0'-def zero-bit0-def
by auto
by auto
show (1::int) < int CARD('a bit0) by (metis bit0.size1)

```
    show (1::int) < int CARD('a bit0) by (metis bit0.size1)
```

```
    show type-definition (Rep::'a bit0 => int) (Abs:: int => 'a bit0) {0::int..<int
CARD('a bit0)}
    proof (unfold type-definition-def Rep-bit0-def [abs-def]
        Abs-bit0-def [abs-def] Abs-bit0'-def, intro conjI)
    show }\forallx::'a bit0. Rep-bit0 x { {0::int..<int CARD('a bit0)
        unfolding card-bit0 unfolding of-nat-mult
        using Rep-bit0 [where ?' }a=\mp@subsup{=}{}{\prime}a]\mathrm{ by simp
    show }\forallx:\mp@subsup{:}{}{\prime}a\mathrm{ bit0. Abs-bit0 (Rep-bit0 x mod int CARD('a bit0)) = x
        by (metis Rep-bit0-inverse bit0.Rep-mod)
    show }\forally::int. y\in{0::int..<int CARD('a bit0)
        Rep-bit0 ((Abs-bit0::int => 'a bit0) (y mod int CARD('a bit0))) = y
        by (metis bit0.Abs-inverse bit0.Rep-mod)
qed
show (1::'a bit0) = Abs (1::int) unfolding Abs-bit0-def Abs-bit0'-def one-bit0-def
    by (metis bit0.of-nat-eq of-nat-1 one-bit0-def)
fix x y :: 'a bit0
show }x+y=Abs((Rep x + Rep y) mod int CARD('a bit0))
    unfolding Abs-bit0-def Rep-bit0-def plus-bit0-def Abs-bit0'-def by fastforce
show }x*y=Abs(Rep x * Rep y mod int CARD('a bit0)
    unfolding Abs-bit0-def Rep-bit0-def times-bit0-def Abs-bit0'-def by fastforce
show }x-y=Abs((Rep x - Rep y) mod int CARD('a bit0))
    unfolding Abs-bit0-def Rep-bit0-def minus-bit0-def Abs-bit0'-def by fastforce
show - x = Abs (- Rep x mod int CARD('a bit0))
    unfolding Abs-bitO-def Rep-bit0-def uminus-bit0-def Abs-bit0'-def by fastforce
show (0::'a bit1) = Abs ( 0::int) unfolding Abs-bit1-def Abs-bit1'-def zero-bit1-def
by auto
    show (1::int) < int CARD('a bit1) by (metis bit1.size1)
    show (1::'a bit1) = Abs (1::int) unfolding Abs-bit1-def Abs-bit1'-def one-bit1-def
    by (metis bit1.of-nat-eq of-nat-1 one-bit1-def)
fix x y :: 'a bit1
show }x+y=Abs((Repx+Rep y) mod int CARD('a bit1))
    unfolding Abs-bit1-def Abs-bit1'-def Rep-bit1-def plus-bit1-def by fastforce
show }x*y=Abs(Rep x*Rep y mod int CARD('a bit1))
    unfolding Abs-bit1-def Rep-bit1-def times-bit1-def Abs-bit1'-def by fastforce
show }x-y=Abs((Repx-Rep y) mod int CARD('a bit1))
    unfolding Abs-bit1-def Rep-bit1-def minus-bit1-def Abs-bit1'-def by fastforce
    show -x = Abs (- Rep x mod int CARD('a bit1))
    unfolding Abs-bit1-def Rep-bit1-def uminus-bit1-def Abs-bit1'-def by fastforce
    show type-definition (Rep::'a bit1 => int) (Abs:: int => 'a bit1) {0::int..<int
CARD('a bit1)}
    proof (unfold type-definition-def Rep-bit1-def [abs-def]
        Abs-bit1-def [abs-def] Abs-bit1'-def, intro conjI)
    have int-2: int 2 = 2 by auto
    show }\forallx::'a bit1. Rep-bit1 x { {0::int..<int CARD('a bit1)
        unfolding card-bit1
        unfolding of-nat-Suc of-nat-mult
        using Rep-bit1 [where ?'a='a]
        unfolding int-2 ..
    show \forallx::'a bit1. Abs-bit1 (Rep-bit1 x mod int CARD('a bit1)) = x
```

```
        by (metis Rep-bit1-inverse bit1.Rep-mod)
    show }\forally::int. y { {0::int..<int CARD('a bit1)
    Rep-bit1 ((Abs-bit1 ::int => 'a bit1) (y mod int CARD('a bit1))) = y
    by (metis bit1.Abs-inverse bit1.Rep-mod)
    qed
    show strict-mono (Rep::'a bit0 => int) unfolding strict-mono-def
    by (metis Rep-bit0-def less-bit0-def)
    show strict-mono (Rep::'a bit1 => int) unfolding strict-mono-def
    by (metis Rep-bit1-def less-bit1-def)
qed
end
end
```


## 3 Miscellaneous

theory Miscellaneous
imports
HOL-Analysis.Determinants
Mod-Type
HOL-Library.Function-Algebras
begin
context Vector-Spaces.linear begin
sublocale vector-space-pair by unfold-locales- TODO: (re)move?
end
hide-const (open) Real-Vector-Spaces.linear
abbreviation linear $\equiv$ Vector-Spaces.linear
In this file, we present some basic definitions and lemmas about linear algebra and matrices.

### 3.1 Definitions of number of rows and columns of a matrix

definition nrows :: ' $a^{\text {^' }}$ columns ${ }^{\text {' }}$ rows $=>$ nat
where nrows $A=C A R D\left({ }^{\prime}\right.$ rows $)$
definition ncols :: ' $a^{\wedge \prime}$ columns ${ }^{\wedge}$ 'rows $=>$ nat
where ncols $A=C A R D$ ('columns)
definition matrix-scalar-mult :: 'a::ab-semigroup-mult $=>^{\prime} a \wedge^{\prime} n \wedge^{\prime} m=>^{\prime} a{ }^{\wedge} n^{\wedge \prime} m$
(infixl $* k$ 70)
where $k * k A \equiv(\chi i j . k * A \$ i \$ j)$

### 3.2 Basic properties about matrices

lemma nrows-not- 0 [simp]:
shows $0 \neq$ nrows $A$ unfolding nrows-def by $\operatorname{simp}$

```
lemma ncols-not-0[simp]:
    shows 0}\not=\mathrm{ ncols A unfolding ncols-def by simp
lemma nrows-transpose: nrows (transpose A) = ncols A
    unfolding nrows-def ncols-def ..
lemma ncols-transpose: ncols (transpose A) = nrows A
    unfolding nrows-def ncols-def ..
lemma finite-rows: finite (rows A)
    using finite-Atleast-Atmost-nat[of \lambdai. row i A] unfolding rows-def .
lemma finite-columns: finite (columns A)
    using finite-Atleast-Atmost-nat[of \lambdai. column i A] unfolding columns-def .
lemma transpose-vector: x v* A = transpose A *vx
    by simp
lemma transpose-zero[simp]:(transpose A=0)=(A=0)
    unfolding transpose-def zero-vec-def vec-eq-iff by auto
```


### 3.3 Theorems obtained from the AFP

The following theorems and definitions have been obtained from the AFP http://isa-afp.org/browser_info/current/HOL/Tarskis_Geometry/Linear Algebra2.html. I have removed some restrictions over the type classes.
lemma vector-scalar-matrix-ac:
fixes $k:: ' a::\{$ field $\}$ and $x:: ' a::\{\text { field }\}^{\wedge} n$ and $A:: '^{\prime} a^{\wedge} m{ }^{\wedge} n$
shows $x v *(k * k A)=k * s(x v * A)$
using scalar-vector-matrix-assoc
unfolding vector-matrix-mult-def matrix-scalar-mult-def vec-eq-iff
by (auto simp add: sum-distrib-left vector-space-over-itself.scale-scale)
lemma transpose-scalar: transpose $(k * k A)=k * k$ transpose $A$
unfolding transpose-def
by (vector, simp add: matrix-scalar-mult-def)
lemma scalar-matrix-vector-assoc:
fixes $A::$ ' $a::\{$ field $\left.\}{ }^{\wedge}\right)^{\wedge}{ }^{\wedge} n$
shows $k * s(A * v v)=k * k A * v v$
by (metis transpose-scalar vector-scalar-matrix-ac vector-transpose-matrix)
lemma matrix-scalar-vector-ac:
fixes $A$ :: ' $a::\{$ field $\}{ }^{\wedge} m^{\wedge} n$
shows $A * v(k * s v)=k * k A * v v$
by (simp add: Miscellaneous.scalar-matrix-vector-assoc vec.scale)

```
definition
    is-basis :: ('a::{field}`'}n) set => bool wher
    is-basis S = vec.independent S^vec.span S=UNIV
lemma card-finite:
    assumes card S = CARD(' n::finite)
    shows finite S
proof -
    from<card S = CARD(' n)> have card S 
    with card-eq-0-iff [of S] show finite S by simp
qed
lemma independent-is-basis:
    fixes B:: ('a::{field} ^'}n) se
    shows vec.independent }B\wedge\mathrm{ card B = CARD('}n)\longleftrightarrow \longleftrightarrows-basis 
proof
    assume vec.independent B ^ card B = CARD('n)
    hence vec.independent B and card B=CARD(' }n)\mathrm{ by simp+
    from card-finite [of B, where ' }n='n]\mathrm{ and <card B=CARD('n)>
    have finite B by simp
    from <card B = CARD(' n)>
    have card B = vec.dim (UNIV :: (('a^'n) set)) unfolding vec-dim-card .
    with vec.card-eq-dim [of B UNIV] and 〈finite B〉 and <vec.independent B〉
    have vec.span B = UNIV by auto
    with 〈vec.independent B〉 show is-basis B unfolding is-basis-def ..
next
    assume is-basis B
    hence vec.independent B unfolding is-basis-def ..
    moreover have card B = CARD('n)
    proof -
        have B\subseteqUNIV by simp
        moreover
        { from\is-basis B> have UNIV \subseteqvec.span B and vec.independent B
            unfolding is-basis-def
                by simp+}
    ultimately have card B = vec.dim (UNIV ::((real^' n) set))
                using vec.basis-card-eq-dim [of B UNIV]
                unfolding vec-dim-card
                by simp
    then show card B = CARD('n)
                by (metis vec-dim-card)
    qed
    ultimately show vec.independent B ^ card B = CARD('n) ..
qed
lemma basis-finite:
    fixes B::('a::{field} `'}n)\mathrm{ set
    assumes is-basis B
    shows finite B
```

```
proof -
    from independent-is-basis [of B] and <is-basis B> have card B = CARD('n)
        by simp
    with card-finite [of B, where ' }n='n\mathrm{ ] show finite }B\mathrm{ by simp
qed
```

Here ends the statements obtained from AFP: http://isa-afp.org/browser info/current/HOL/Tarskis_Geometry/Linear_Algebra2.html which have been generalized.

### 3.4 Basic properties involving span, linearity and dimensions

context finite-dimensional-vector-space
begin
This theorem is the reciprocal theorem of local.independent ? $B \Longrightarrow$ finite ? $B \wedge$ card ? $B=$ local.dim (local.span ? B)
lemma card-eq-dim-span-indep:
assumes $\operatorname{dim}(\operatorname{span} A)=\operatorname{card} A$ and finite $A$
shows independent $A$
by (metis assms card-le-dim-spanning dim-subset equalityE span-superset)
lemma dim-zero-eq:
assumes $\operatorname{dim}-A: \operatorname{dim} A=0$
shows $A=\{ \} \vee A=\{0\}$
using dim-A local.card-ge-dim-independent local.independent-empty by force
lemma dim-zero-eq':
assumes $A: A=\{ \} \vee A=\{0\}$
shows $\operatorname{dim} A=0$
using assms local.dim-span local.indep-card-eq-dim-span local.independent-empty by fastforce
lemma dim-zero-subspace-eq:
assumes subs- $A$ : subspace $A$
shows $(\operatorname{dim} A=0)=(A=\{0\})$
by (metis dim-zero-eq dim-zero-eq' subspace- $0[O F$ subs-A] empty-iff)
lemma span-0-imp-set-empty-or-0:
assumes span $A=\{0\}$
shows $A=\{ \} \vee A=\{0\}$ by (metis assms span-superset subset-singletonD)
end
context Vector-Spaces.linear
begin
lemma linear-injective-ker-0:
shows $\operatorname{inj} f=(\{x . f x=0\}=\{0\})$
using inj-iff-eq-0 by auto
end
lemma snd-if-conv:
shows snd (if $P$ then $(A, B)$ else $(C, D))=($ if $P$ then $B$ else $D)$ by simp

### 3.5 Basic properties about matrix multiplication

lemma row-matrix-matrix-mult:
fixes $A::^{\prime} a::\{\text { comm-ring-1 }\}^{\wedge} n^{\wedge \prime} m$
shows $(P \$ i) v * A=(P * * A) \$ i$
unfolding vec-eq-iff
unfolding vector-matrix-mult-def unfolding matrix-matrix-mult-def
by (auto intro!: sum.cong)
corollary row-matrix-matrix-mult':
fixes $A::^{\prime} a::\{\text { comm-ring-1 }\}^{\wedge} n^{\wedge \prime} m$
shows (row $i P$ ) v* $A=$ row $i(P * * A)$
using row-matrix-matrix-mult unfolding row-def vec-nth-inverse .
lemma column-matrix-matrix-mult:
shows column $i(P * * A)=P * v($ column $i A)$
unfolding column-def matrix-vector-mult-def matrix-matrix-mult-def by fastforce
lemma matrix-matrix-mult-inner-mult:
shows $(A * * B) \$ i \$ j=$ row $i A \cdot$ column $j B$
unfolding inner-vec-def matrix-matrix-mult-def row-def column-def by auto
lemma matrix-vmult-column-sum:
fixes $A::^{\prime} a::\{\text { field }\}^{\wedge} n^{\wedge} n^{\prime} m$
shows $\exists f . A * v x=\operatorname{sum}(\lambda y . f y * s y)($ columns $A)$
proof (rule exI[of - $\lambda y$. sum ( $\lambda i . x \$ i$ ) $\{i . y=$ column $i A\}])$
let ? $f=\lambda y$. sum $(\lambda i . x \$ i)\{i . y=$ column $i A\}$
let $? g=(\lambda y .\{i . y=$ column $i(A)\})$
have inj: inj-on ?g (columns $(A)$ ) unfolding inj-on-def unfolding columns-def by auto
have union-univ: $\bigcup\left(? g^{\prime}(\right.$ columns $\left.(A))\right)=U N I V$ unfolding columns-def by auto
have $A * v x=\left(\sum i \in U N I V . x \$ i * s\right.$ column $\left.i A\right)$ unfolding matrix-mult-sum
..
also have $\ldots=\operatorname{sum}(\lambda i . x \$ i * s$ column $i A)\left(\bigcup\left(? g^{\prime}(\right.\right.$ columns $\left.\left.A)\right)\right)$ unfolding
union-univ ..
also have $\ldots=\operatorname{sum}(\operatorname{sum}((\lambda i . x \$ i * s$ column $i A)))\left(? g^{\prime}(\right.$ columns $\left.A)\right)$
by (rule sum.Union-disjoint[unfolded o-def], auto)
also have $\ldots=\operatorname{sum}((\operatorname{sum}((\lambda i . \quad x \$ i * s$ column $i A))) \circ$ ? $g) \quad($ columns $A)$
by (rule sum.reindex, simp add: inj)
also have $\ldots=\operatorname{sum}(\lambda y$. ?f $y * s y)($ columns $A)$

```
proof (rule sum.cong, unfold o-def)
    fix \(x a\)
    have \(\operatorname{sum}(\lambda i . x \$ i * s\) column \(i A)\{i . x a=\) column \(i A\}\)
        \(=\operatorname{sum}(\lambda i . x \$ i * s x a)\{i\). xa \(=\) column \(i A\}\) by simp
    also have \(\ldots=\operatorname{sum}(\lambda i . x \$ i)\{i . x a=\) column \(i A\} * s x a\)
            using vec.scale-sum-left[of ( \(\lambda i . x \$ i)\{i . x a=\) column \(i A\} x a]\).
    finally show \(\left(\sum i \mid x a=\right.\) column \(i A . x \$ i * s\) column \(\left.i A\right)=\left(\sum i \mid x a=\right.\)
column i A. \(x \$ i) * s x a\).
    qed rule
    finally show \(A * v x=\left(\sum y \in\right.\) columns \(A .\left(\sum i \mid y=\right.\) column \(\left.\left.i A . x \$ i\right) * s y\right)\).
qed
```


### 3.6 Properties about invertibility

```
lemma matrix-inv:
    assumes invertible M
    shows matrix-inv-left: matrix-inv M** M = mat 1
    and matrix-inv-right: M** matrix-inv M = mat 1
    using <invertible M` and someI-ex [of \lambdaN.M**N=mat 1 ^N**M=mat
1]
    unfolding invertible-def and matrix-inv-def
    by simp-all
```

In the library, matrix-inv ? $A=\left(S O M E A^{\prime}\right.$. ? $A * * A^{\prime}=m a t\left(1:: ?^{\prime} a\right) \wedge A^{\prime}$
** ? $A=$ mat $\left.\left(1:: ?^{\prime} a\right)\right)$ allows the use of non squary matrices. The following
lemma can be also proved fixing $A$
lemma matrix-inv-unique:
fixes $A::^{\prime} a::\{\text { semiring }-1\}^{\wedge} n{ }^{\wedge} n$
assumes $A B: A * * B=$ mat 1 and $B A: B * * A=$ mat 1
shows matrix-inv $A=B$
by (metis $A B$ BA invertible-def matrix-inv-right matrix-mul-assoc matrix-mul-lid)
lemma matrix-vector-mult-zero-eq:
assumes $P$ : invertible $P$
shows $((P * * A) * v x=0)=(A * v x=0)$
proof (rule iffI)
assume $P * * A * v x=0$
hence matrix-inv $P * v(P * * A * v x)=$ matrix-inv $P * v 0$ by simp
hence matrix-inv $P * v(P * * A * v x)=0$ by (metis matrix-vector-mult- 0 -right)
hence (matrix-inv $P * * P * * A) * v x=0$ by (metis matrix-vector-mul-assoc)
thus $A * v x=0$ by (metis assms matrix-inv-left matrix-mul-lid)
next
assume $A * v x=0$
thus $P * * A * v x=0$ by (metis matrix-vector-mul-assoc matrix-vector-mult-0-right)
qed
lemma independent-image-matrix-vector-mult:
fixes $P::{ }^{\prime} a::\{$ fiel $d\}{ }^{\prime} n^{\wedge}{ }^{\prime} m$
assumes ind-B: vec.independent $B$ and inv- $P$ : invertible $P$
shows vec.independent $\left(((* v) P)^{‘} B\right)$
proof (rule vec.independent-injective-image)
show vec.independent $B$ using ind- $B$.
show inj-on ( $(* v) P$ ) (vec.span $B$ )
using inj-matrix-vector-mult $[$ [OF inv-P] unfolding inj-on-def by simp
qed
lemma independent-preimage-matrix-vector-mult:
fixes $P::{ }^{\prime} a::\{$ field $\}{ }^{\wedge} n^{\wedge} n$
assumes ind-B: vec.independent $\left(((* v) P)^{‘} B\right)$ and inv-P: invertible $P$
shows vec.independent $B$
proof -
have vec.independent $\left(((* v)(\text { matrix-inv } P))^{‘}\left(((* v) P)^{〔} B\right)\right)$
proof (rule independent-image-matrix-vector-mult)
show vec.independent $((* v) P$ ‘ $B$ ) using ind- $B$.
show invertible (matrix-inv $P$ )
by (metis matrix-inv-left matrix-inv-right inv-P invertible-def)
qed
moreover have $((* v)$ (matrix-inv $P))^{`}\left(((* v) P)^{`} B\right)=B$
proof (auto)
fix $x$ assume $x: x \in B$ show matrix-inv $P * v(P * v x) \in B$
by (metis (full-types) x inv-P matrix-inv-left matrix-vector-mul-assoc ma-
trix-vector-mul-lid)
thus $x \in(* v)($ matrix-inv $P)$ ' $(* v) P$ ' $B$
unfolding image-def
by (auto, metis inv-P matrix-inv-left matrix-vector-mul-assoc matrix-vector-mul-lid)
qed
ultimately show ?thesis by simp
qed

### 3.7 Properties about the dimension of vectors

lemma dimension-vector[code-unfold]: vec.dimension TYPE('a::\{\{ield\}) TYPE('rows:::\{mod-type\})=CARD('
proof -
let $? f=\lambda x$. axis (from-nat $x$ ) $1::^{\prime} a^{\wedge}$ rows:: $\{$ mod-type $\}$
have vec.dimension TYPE ('a:::\{field\}) TYPE ('rows::\{mod-type\}) $=$ card (cart-basis::('a^'rows::\{mod-type\})
set)
unfolding vec.dimension-def ..
also have $\ldots=\operatorname{card}\{. .<C A R D($ 'rows $)\}$ unfolding cart-basis-def
proof (rule bij-betw-same-card[symmetric, of ?f], unfold bij-betw-def, unfold
inj-on-def axis-eq-axis, auto)
fix $x y$ assume $x: x<\operatorname{CARD}$ ('rows) and $y: y<\operatorname{CARD}$ ('rows) and eq:
from-nat $x=($ from-nat $y::$ 'rows $)$
show $x=y$ using from-nat-eq-imp-eq $[$ OF eq $x y]$.
next
fix $i$ show axis i $1 \in(\lambda x$. axis (from-nat x::'rows) 1)' $\{. .<\operatorname{CARD}($ 'rows $)\}$
unfolding image-def

```
    by (auto, metis lessThan-iff to-nat-from-nat to-nat-less-card)
    qed
also have ... = CARD('rows) by (metis card-lessThan)
finally show ?thesis.
qed
```


### 3.8 Instantiations and interpretations

Functions between two real vector spaces form a real vector
instantiation fun :: (real-vector, real-vector) real-vector
begin
definition scaleR-fun a $f=\left(\lambda i . a *_{R} f i\right)$
instance
by (intro-classes, auto simp add: fun-eq-iff scaleR-fun-def scaleR-left.add scaleR-right.add) end

```
instantiation vec :: (type, finite) equal
begin
definition equal-vec \(::\left({ }^{\prime} a\right.\), 'b::finite) vec \(=>\left({ }^{\prime} a\right.\), ' \(b::\) finite) vec \(=>b o o l\)
    where equal-vec \(x y=(\forall i . x \$ i=y \$ i)\)
instance
proof (intro-classes)
    fix \(x y::(' a\), ' \(b::\) finite) \(v e c\)
    show equal-class.equal \(x y=(x=y)\) unfolding equal-vec-def using vec-eq-iff
by auto
qed
end
interpretation matrix: vector-space \(((* k)):::^{\prime} a::\{\) field \(\}=>^{\prime} a{ }^{\wedge}\) cols \({ }^{\wedge}\) rows \(=>^{\prime} a^{\wedge^{\prime}}\) cols \({ }^{\wedge}\) rows
proof (unfold-locales)
fix \(a::^{\prime} a\) and \(x y::^{\prime} a^{\wedge \prime} c^{\prime}{ }^{\wedge}\) 'rows
show \(a * k(x+y)=a * k x+a * k y\)
    unfolding matrix-scalar-mult-def vec-eq-iff
    by (simp add: vector-space-over-itself.scale-right-distrib)
next
fix \(a b::^{\prime} a\) and \(x::^{\prime} a^{\wedge \prime}\) cols \({ }^{\wedge}\) rows
show \((a+b) * k x=a * k x+b * k x\)
unfolding matrix-scalar-mult-def vec-eq-iff
    by (simp add: comm-semiring-class.distrib)
show \(a * k(b * k x)=a * b * k x\)
    unfolding matrix-scalar-mult-def vec-eq-iff by auto
show \(1 * k x=x\) unfolding matrix-scalar-mult-def vec-eq-iff by auto
qed
end
```


## 4 Fundamental Subspaces

theory Fundamental-Subspaces<br>imports<br>Miscellaneous<br>begin

### 4.1 The fundamental subspaces of a matrix

### 4.1.1 Definitions

definition left-null-space :: 'a::\{semiring-1 $\}^{\wedge} n^{\wedge \prime} m=>\left({ }^{\prime} a^{\wedge \prime} m\right)$ set where left-null-space $A=\{x$. x $v * A=0\}$
definition null-space $::$ ' $a::\{\text { semiring- } 1\}^{\wedge \prime} n n^{\prime} m=>\left({ }^{\prime} a^{\wedge} n\right)$ set where null-space $A=\{x . A * v x=0\}$
definition row-space $::$ ' $a::\{$ field $\}{ }^{\wedge} n^{\wedge} \quad m=>\left({ }^{\prime} a^{\wedge} n\right)$ set where row-space $A=$ vec.span (rows $A$ )
definition col-space :: ' $a::\{$ field $\}{ }^{\wedge} n n^{\prime \prime} m=>\left({ }^{\prime} a^{\prime \prime} m\right)$ set
where col-space $A=$ vec.span (columns $A$ )

### 4.1.2 Relationships among them

lemma left-null-space-eq-null-space-transpose: left-null-space $A=$ null-space (transpose A)
unfolding null-space-def left-null-space-def transpose-vector ..
lemma null-space-eq-left-null-space-transpose: null-space $A=$ left-null-space (transpose A)
using left-null-space-eq-null-space-transpose[of transpose A] unfolding transpose-transpose ..
lemma row-space-eq-col-space-transpose:
fixes $A::^{\prime} a::\{$ field $\}{ }^{\wedge}$ 'columns ${ }^{\wedge}$ 'rows shows row-space $A=$ col-space (transpose $A$ ) unfolding col-space-def row-space-def columns-transpose[of A] ..
lemma col-space-eq-row-space-transpose:
fixes $A::^{\prime} a::\{\text { field }\}^{\wedge \prime} n^{\wedge \prime} m$
shows col-space $A=$ row-space (transpose $A$ )
unfolding col-space-def row-space-def unfolding rows-transpose[of A] ..

### 4.2 Proving that they are subspaces

```
lemma subspace-null-space:
    fixes \(A:::^{\prime} a:\{\text { field }\}^{\wedge} n^{\wedge}{ }^{\wedge} m\)
    shows vec.subspace (null-space A)
    by (auto simp: vec.subspace-def null-space-def vec.scale vec.add)
```

```
lemma subspace-left-null-space:
    fixes A::' a::{field} `' }n\mp@subsup{}{}{\wedge\prime}
    shows vec.subspace (left-null-space A)
    unfolding left-null-space-eq-null-space-transpose using subspace-null-space .
lemma subspace-row-space:
    shows vec.subspace (row-space A) by (metis row-space-def vec.subspace-span)
lemma subspace-col-space:
    shows vec.subspace (col-space A) by (metis col-space-def vec.subspace-span)
```


### 4.3 More useful properties and equivalences

lemma col-space-eq:
fixes $A::^{\prime} a::\{\text { field }\}^{\wedge \prime} m::\{\text { finite, wellorder }\}^{\wedge \prime} n$
shows col-space $A=\{y . \exists x . A * v x=y\}$
proof (unfold col-space-def vec.span-finite[OF finite-columns], auto)
fix $x$
show $A * v x \in$ range $\left(\lambda u . \sum v \in\right.$ columns $\left.A . u v * s v\right)$ using matrix-vmult-column-sum $[$ of
$A x]$ by auto
next
fix $u::\left({ }^{\prime} a,{ }^{\prime} n\right)$ vec $\Rightarrow{ }^{\prime} a$
let $? g=\lambda y$. $\{i . y=$ column $i A\}$
let ? $x=(\chi$. if $i=($ LEAST $a . a \in$ ? $g($ column $i A))$ then $u($ column $i A)$ else 0)
show $\exists x . A * v x=\left(\sum v \in\right.$ columns $\left.A . u v * s v\right)$
proof (unfold matrix-mult-sum, rule exI[of - ? $x$ ], auto)
have inj: inj-on ?g (columns A) unfolding inj-on-def unfolding columns-def by auto
have union-univ: $\bigcup\left(? g^{\prime}(\right.$ columns $\left.A)\right)=U N I V$ unfolding columns-def by auto
have sum $(\lambda i$. (if $i=($ LEAST $a$. column $i A=$ column a $A)$ then $u($ column $i$ A) else 0) $*$ s column $i$ A) UNIV
$=\operatorname{sum}(\lambda i$. (if $i=($ LEAST $a$. column $i A=$ column a A) then $u($ column $i$
A) else 0) *s column $i A)\left(\bigcup\left(? g^{\prime}(\right.\right.$ columns $\left.\left.A)\right)\right)$
unfolding union-univ ..
also have $\ldots=\operatorname{sum}(\operatorname{sum}(\lambda i .($ if $i=($ LEAST a. column $i A=$ column a $A)$
then $u($ column $i A)$ else 0$) * s$ column $i A))\left(? g^{\prime}(\right.$ columns $\left.A)\right)$
by (rule sum.Union-disjoint[unfolded o-def], auto)
also have $\ldots=\operatorname{sum}((\operatorname{sum}(\lambda i .($ if $i=(L E A S T$ a. column i $A=$ column a $A)$
then $u($ column $i A)$ else 0$) * s$ column $i A)) \circ$ ? $g)$
(columns A) by (rule sum.reindex, simp add: inj)
also have $\ldots=\operatorname{sum}(\lambda y . u y * s y)($ columns $A)$
proof (rule sum.cong, auto)
fix $x$
assume $x$-in-cols: $x \in$ columns $A$
obtain $b$ where $b$ : $x=$ column $b A$ using $x$-in-cols unfolding columns-def by blast
let ? $f=(\lambda i$. (if $i=($ LEAST $a$. column $i A=$ column a $A)$ then $u($ column $i$
A) else 0) $*$ s column i A)
have sum-rw: sum ?f $(\{i . x=$ column $i A\}-\{\operatorname{LEAST}$ a. $x=$ column a $A\})$ $=0$
by (rule sum.neutral, auto)
have sum ?f $\{i . x=$ column $i A\}=$ ?f (LEAST a. $x=$ column a $A)+$ sum ?f $(\{i . x=$ column $i A\}-\{L E A S T$ a. $x=$ column a $A\})$
apply (rule sum.remove, auto, rule LeastI-ex)
using $x$-in-cols unfolding columns-def by auto
also have $\ldots=$ ?f (LEAST $a . x=$ column $a$ A) unfolding sum-rw by simp also have $\ldots=u x * s x$ proof (auto, rule LeastI2)
show $x=$ column $b A$ using $b$.
fix $x a$
assume $x: x=$ column $x a A$
show $u($ column xa $A$ ) $* s$ column xa $A=u x * s x$ unfolding $x$..
next
assume $(L E A S T$ a. $x=$ column a $A) \neq($ LEAST a. column (LEAST c. $x$ $=$ column с A) $A=$ column a A)
moreover have $(L E A S T$ a. $x=$ column a $A)=($ LEAST a. column $(L E A S T$ c. $x=$ column c $A$ ) $A=$ column a $A$ )
by (rule Least-equality[symmetric], rule LeastI2, simp-all add: b, rule Least-le, metis (lifting, full-types) LeastI)
ultimately show $u x=0$ by contradiction
qed
finally show ( $\sum i \mid x=$ column $i A$. (if $i=(L E A S T$ a. column $i A=$ column a A) then $u($ column $i A)$ else 0) $* s$ column $i A)=u x * s x$.
qed
finally show ( $\sum i \in U N I V$. (if $i=(L E A S T$ a. column i $A=$ column a $A$ ) then $u($ column $i A)$ else 0$) * s$ column i $A)=\left(\sum y \in\right.$ columns $\left.A . u y * s y\right)$. qed
qed
corollary col-space-eq':
fixes $A::^{\prime} a::\{$ field $\}{ }^{\wedge \prime} m::\{\text { finite, wellorder }\}^{\wedge \prime} n$
shows col-space $A=$ range $(\lambda x . A * v x)$
unfolding col-space-eq by auto
lemma row-space-eq:
fixes $A:: ' a::\{$ field $\}{ }^{\wedge} m^{\wedge \prime} n::\{$ finite, wellorder $\}$
shows row-space $A=\{w . \exists y$. (transpose $A) * v y=w\}$
unfolding row-space-eq-col-space-transpose col-space-eq ..
lemma null-space-eq-ker
fixes $f::\left({ }^{\prime} a:: f i e l d{ }^{\wedge} \prime n\right)=>\left({ }^{\prime}{ }^{\wedge} / m\right)$
assumes lf: Vector-Spaces.linear $(* s)(* s) f$
shows null-space $($ matrix $f)=\{x . f x=0\}$
unfolding null-space-def using matrix-works $[O F l f]$ by auto

```
lemma col-space-eq-range:
    fixes f::('a::field^'}n::{\mathrm{ finite, wellorder }) }=>('a\mp@subsup{}{}{\wedge\prime}m
    assumes lf:Vector-Spaces.linear (*s) (*s)f
    shows col-space (matrix f) = range f
    unfolding col-space-eq unfolding matrix-works[OF lf] by blast
lemma null-space-is-preserved:
    fixes A::'a::{field} ^'cols`'rows
    assumes P: invertible P
    shows null-space ( }P**A)=\mathrm{ null-space A
    unfolding null-space-def
    using P matrix-inv-left matrix-left-invertible-ker matrix-vector-mul-assoc ma-
trix-vector-mult-0-right
    by metis
lemma row-space-is-preserved:
    fixes }A::'a::{field}^'cols`'rows::{finite, wellorder
        and P::'a::{field}^'rows::{finite, wellorder}^'rows::{finite, wellorder}
    assumes P}P\mathrm{ : invertible }
    shows row-space ( }P**A)=\mathrm{ row-space A
proof (auto)
    fix }
    assume w: w\in row-space (P**A)
    from this obtain }y\mathrm{ where w-By:w=(transpose (P**A))*vy
        unfolding row-space-eq[of P ** A ] by fast
    have w}=(\mathrm{ transpose ( }P**A))*vy\mathrm{ using w-By .
    also have }\ldots=((\mathrm{ transpose }A)**(\mathrm{ transpose P)) *v y unfolding matrix-transpose-mul
    also have ... = (transpose A)*v((transpose P)*vy) unfolding matrix-vector-mul-assoc
finally show w}\mathrm{ (row-space A unfolding row-space-eq by blast
next
    fix w
    assume w: w\in row-space A
    from this obtain y where w-Ay:w=(transpose A) *v y unfolding row-space-eq
by fast
    have w=(transpose A)*v y using w-Ay .
    also have ... = (transpose ((matrix-inv P) ** (P**A))) *v y
        by (metis P matrix-inv-left matrix-mul-assoc matrix-mul-lid)
    also have ... = (transpose (P**A) ** (transpose (matrix-inv P))) *v y
        unfolding matrix-transpose-mul ..
    also have ... = transpose ( }P**A)*v(\mathrm{ transpose (matrix-inv P)*v y)
        unfolding matrix-vector-mul-assoc ..
    finally show w}\mathrm{ frow-space ( }P**A)\mathrm{ unfolding row-space-eq by blast
qed
end
```


## 5 Rank Nullity Theorem of Linear Algebra

```
theory Dim-Formula
    imports Fundamental-Subspaces
begin
context vector-space
begin
```


### 5.1 Previous results

Linear dependency is a monotone property, based on the monotonocity of linear independence:

```
lemma dependent-mono:
    assumes d:dependent A
    and A-in-B: A\subseteqB
    shows dependent B
    using independent-mono [OF-A-in-B] d by auto
```

Given a finite independent set, a linear combination of its elements equal to zero is possible only if every coefficient is zero:
lemma scalars-zero-if-independent:
assumes fin-A: finite $A$
and ind: independent $A$
and sum: $\left(\sum x \in A\right.$. scale $\left.(f x) x\right)=0$
shows $\forall x \in A . f x=0$
using fin- $A$ ind local.dependent-finite sum by blast
end
context finite-dimensional-vector-space
begin
In an finite dimensional vector space, every independent set is finite, and thus

【finite $A$; local.independent $A ;\left(\sum x \in A . f x * s x\right)=\left(0::{ }^{\prime} b\right) \rrbracket$
$\Longrightarrow \forall x \in A . f x=\left(0::^{\prime} a\right)$
holds:
corollary scalars-zero-if-independent-euclidean:
assumes ind: independent $A$
and sum: $\left(\sum x \in A\right.$. scale $\left.(f x) x\right)=0$
shows $\forall x \in A$. $f x=0$
using finiteI-independent ind scalars-zero-if-independent sum by blast
end

The following lemma states that every linear form is injective over the elements which define the basis of the range of the linear form. This property is applied later over the elements of an arbitrary basis which are not in the basis of the nullifier or kernel set (i.e., the candidates to be the basis of the range space of the linear form).

Thanks to this result, it can be concluded that the cardinal of the elements of a basis which do not belong to the kernel of a linear form $f$ is equal to the cardinal of the set obtained when applying $f$ to such elements.

The application of this lemma is not usually found in the pencil and paper proofs of the "rank nullity theorem", but will be crucial to know that, being $f$ a linear form from a finite dimensional vector space $V$ to a vector space $V^{\prime}$, and given a basis $B$ of $\operatorname{ker} f$, when $B$ is completed up to a basis of $V$ with a set $W$, the cardinal of this set is equal to the cardinal of its range set:

```
context vector-space
begin
lemma inj-on-extended:
    assumes lf:Vector-Spaces.linear scaleB scaleC f
    and f: finite C
    and ind-C: independent C
    and C-eq:C=B\cupW
    and disj-set: }B\capW={
    and span-B: {x.fx=0}\subseteq span B
    shows inj-on f W
```

    - The proof is carried out by reductio ad absurdum
    proof (unfold inj-on-def, rule+, rule ccontr)
interpret lf: Vector-Spaces.linear scaleB scaleC $f$ using lf by simp
- Some previous consequences of the premises that are used later:
have fin-B: finite $B$ using finite-subset $[O F-f] C$-eq by simp
have ind- $B$ : independent $B$ and ind- $W$ : independent $W$
using independent-mono[OF ind-C] C-eq by simp-all
- The proof starts here; we assume that there exist two different elements
- with the same image:
fix $x::^{\prime} b$ and $y::^{\prime} b$
assume $x: x \in W$ and $y: y \in W$ and $f$-eq: $f x=f y$ and $x$-not- $y: x \neq y$
have fin-yB: finite (insert y $B$ ) using fin- $B$ by simp
have $f(x-y)=0$ by (metis diff-self $f$-eq lf.diff)
hence $x-y \in\{x . f x=0\}$ by simp
hence $\exists g$. $\left(\sum v \in B\right.$. scale $\left.\binom{g}{v} v\right)=(x-y)$ using span- $B$
unfolding span-finite $[O F$ fin- $B]$ by force
then obtain $g$ where sum: $\left(\sum v \in B\right.$. scale $\left.(g v) v\right)=(x-y)$ by blast
- We define one of the elements as a linear combination of the second element
and the ones in $B$
define $h::{ }^{\prime} b \Rightarrow{ }^{\prime} a$ where $h a=($ if $a=y$ then 1 else $g a$ ) for $a$
have $x=y+\left(\sum v \in B\right.$. scale $\left.(g v) v\right)$ using sum by auto
also have $\ldots=$ scale $(h y) y+\left(\sum v \in B\right.$. scale $\left.(g v) v\right)$ unfolding $h$-def by simp

```
also have \(\ldots=\) scale ( \(h y\) ) \(y+\left(\sum v \in B\right.\). scale ( \(h v\) ) v)
    apply (unfold add-left-cancel, rule sum.cong)
    using \(y\) h-def empty-iff disj-set by auto
also have \(\ldots=\left(\sum v \in(\right.\) insert \(y B)\). scale \(\left.(h v) v\right)\)
    by (rule sum.insert[symmetric], rule fin-B)
        (metis (lifting) IntI disj-set empty-iff y)
    finally have \(x\)-in-span-yB: \(x \in\) span (insert \(y B\) )
    unfolding span-finite \([O F\) fin-yB] by auto
    - We have that a subset of elements of \(C\) is linearly dependent
    have dep: dependent (insert \(x\) (insert y B))
    by (unfold dependent-def, rule bexI [of - x])
        (metis Diff-insert-absorb Int-iff disj-set empty-iff insert-iff
                x x-in-span-y \(B\) x-not- \(y\), simp)
- Therefore, the set \(C\) is also dependent:
    hence dependent \(C\) using \(C\)-eq \(x\) y
    by (metis Un-commute Un-upper2 dependent-mono insert-absorb insert-subset)
    - This yields the contradiction, since \(C\) is independent:
    thus False using ind- \(C\) by contradiction
qed
end
```


### 5.2 The proof

Now the rank nullity theorem can be proved; given any linear form $f$, the sum of the dimensions of its kernel and range subspaces is equal to the dimension of the source vector space.

The statement of the "rank nullity theorem for linear algebra", as well as its proof, follow the ones on [1]. The proof is the traditional one found in the literature. The theorem is also named "fundamental theorem of linear algebra" in some texts (for instance, in [2]).
context finite-dimensional-vector-space
begin
theorem rank-nullity-theorem:
assumes l: Vector-Spaces.linear scale scaleC $f$
shows dimension $=\operatorname{dim}\{x . f x=0\}+$ vector-space.dim scale $C$ (range $f$ )
proof -

- For convenience we define abbreviations for the universe set, $V$, and the kernel of $f$
interpret l: Vector-Spaces.linear scale scaleC $f$ by fact
define $V$ :: ' $b$ set where $V=U N I V$
define ker-f where ker-f $=\{x . f x=0\}$
- The kernel is a proper subspace:
have sub-ker: subspace $\{x . f x=0\}$ using l.subspace-kernel.
- The kernel has its proper basis, $B$ :
obtain $B$ where $B$-in-ker: $B \subseteq\{x . f x=0\}$
and independent- $B$ : independent $B$
and ker-in-span: $\{x . f x=0\} \subseteq \operatorname{span} B$
and card-B: card $B=\operatorname{dim}\{x . f x=0\}$ using basis-exists by blast
- The space $V$ has a (finite dimensional) basis, $C$ :
obtain $C$ where $B$-in- $C: B \subseteq C$ and $C$-in- $V: C \subseteq V$
and independent- $C$ : independent $C$
and span- $C$ : $V=$ span $C$
unfolding $V$-def
by (metis independent-B extend-basis-superset independent-extend-basis span-extend-basis span-superset)
- The basis of $V, C$, can be decomposed in the disjoint union of the basis of the kernel, $B$, and its complementary set, $C-B$
have $C$-eq: $C=B \cup(C-B)$ by (rule Diff-partition [OF B-in- $C$, symmetric])
have $e q-f C$ : $f^{\prime} C=f^{\prime} B \cup f^{\prime}(C-B)$
by (subst $C$-eq, unfold image-Un, simp)
- The basis $C$, and its image, are finite, since $V$ is finite-dimensional
have finite- $C$ : finite $C$
using finiteI-independent $[O F$ independent- $C]$.
have finite-fC: finite ( $f^{\prime} C$ ) by (rule finite-imageI $[O F$ finite- $C]$ )
- The basis $B$ of the kernel of $f$, and its image, are also finite
have finite-B: finite $B$ by (rule rev-finite-subset [OF finite-C B-in-C])
have finite-fB: finite $\left(f^{\prime} B\right)$ by (rule finite-imageI $[O F$ finite- $B]$ )
- The set $C-B$ is also finite
have finite-CB: finite $(C-B)$ by (rule finite-Diff [OF finite- $C$, of $B]$ )
have dim-ker-le-dim-V:dim $(k e r-f) \leq \operatorname{dim} V$
using dim-subset [of ker-f $V$ ] unfolding $V$-def by simp
- Here it starts the proof of the theorem: the sets $B$ and $C-B$ must be proven to be bases, respectively, of the kernel of $f$ and its range
show ?thesis
proof -
have dimension $=\operatorname{dim} V$ unfolding $V$-def dim-UNIV dimension-def
by (metis basis-card-eq-dim dimension-def independent-Basis span-Basis top-greatest)
also have $\operatorname{dim} V=\operatorname{dim} C$ unfolding span- $C$ dim-span ..
also have $\ldots=$ card $C$
using basis-card-eq-dim [of C $C$, OF - span-superset independent- $C$ ] by simp
also have $\ldots=\operatorname{card}(B \cup(C-B))$ using $C$-eq by simp
also have $\ldots=\operatorname{card} B+\operatorname{card}(C-B)$
by (rule card-Un-disjoint[OF finite-B finite-CB], fast)
also have $\ldots=\operatorname{dim} k e r-f+\operatorname{card}(C-B)$ unfolding ker-f-def card- $B$..
- Now it has to be proved that the elements of $C-B$ are a basis of the range of $f$
also have..$=\operatorname{dim} \operatorname{ker}-f+l . v s 2 . \operatorname{dim}($ range $f)$
proof (unfold add-left-cancel)
define $W$ where $W=C-B$
have finite- $W$ : finite $W$ unfolding $W$-def using finite- $C B$.
have finite-f $W$ : finite $(f$ ' $W$ ) using finite-imageI $[O F$ finite- $W]$.
have card $W=\operatorname{card}\left(f^{\prime} W\right)$
by (rule card-image [symmetric], rule inj-on-extended $[$ OF $l$, of $C$ B], rule finite- $C$ )
(rule independent-C, unfold $W$-def, subst $C$-eq, rule refl, simp, rule ker-in-span)
also have $\ldots=$ l.vs2. $\operatorname{dim}($ range $f)$
- The image set of $W$ is independent and its span contains the range of $f$, so it is a basis of the range:
proof (rule l.vs2.basis-card-eq-dim)
- 1. The image set of $W$ generates the range of $f$ :
show range $f \subseteq$ l.vs2.span $(f$ ' $W$ )
proof (unfold l.vs2.span-finite [OF finite-fW], auto)
- Given any element $v$ in $V$, its image can be expressed as a linear combination of elements of the image by $f$ of $C$ :
fix $v:: \quad b$
have $f V$-span: $f^{\prime} V \subseteq l . v s 2 . s p a n ~(f$ ' $C)$
by (simp add: span-C l.span-image)
have $\exists g$. $\left(\sum x \in f^{f} C\right.$. scale $\left.C(g x) x\right)=f v$
using $f V$-span unfolding $V$-def
using l.vs2.span-finite[OF finite-fC]
by (metis (no-types, lifting) V-def rangeE rangeI span-C l.span-image)
then obtain $g$ where $f v: f v=\left(\sum x \in f ' C\right.$. scale $\left.C(g x) x\right)$ by metis
- We recall that $C$ is equal to $B$ union $(C-B)$, and $B$ is the basis of the kernel; thus, the image of the elements of $B$ will be equal to zero:
have zero-fB: $\left(\sum x \in f\right.$ ' $B$. scale $\left.C(g x) x\right)=0$
using $B$-in-ker by (auto intro!: sum.neutral)
have zero-inter: $\left(\sum x \in\left(f^{\prime} B \cap f^{\prime} W\right)\right.$. scale $\left.C(g x) x\right)=0$ using $B$-in-ker by (auto intro!: sum.neutral)
have $f v=\left(\sum x \in f^{\prime} C\right.$. scale $\left.C(g x) x\right)$ using $f v$.
also have $\ldots=\left(\sum x \in\left(f\right.\right.$ ' $\left.B \cup f^{\prime} W\right)$. scaleC $\left.(g x) x\right)$ using eq-fC $W$-def by simp
also have...$=$

$$
\begin{aligned}
& \left(\sum x \in f^{‘} B . \text { scale } C(g x) x\right)+\left(\sum x \in f \text { ‘} W . \text { scale } C(g x) x\right) \\
& \quad-\left(\sum x \in\left(f^{\prime} B \cap f^{\prime} W\right) . \text { scaleC }(g x) x\right)
\end{aligned}
$$

using sum-Un [OF finite-fB finite-fW] by simp
also have $\ldots=\left(\sum x \in f^{\prime} W\right.$. scale $\left.C(g x) x\right)$
unfolding zero-fB zero-inter by simp

- We have proved that the image set of $W$ is a generating set of the
range of $f$
finally show $f v \in$ range $\left(\lambda u . \sum v \in f\right.$ ' $W$. scale $\left.C(u v) v\right)$ by auto
qed
- 2. The image set of $W$ is linearly independent:
show l.vs2.independent ( $f$ ' $W$ )
using finite-f $W$
proof (rule l.vs2.independent-if-scalars-zero)
- Every linear combination (given by $g x$ ) of the elements of the image set of $W$ equal to zero, requires every coefficient to be zero:
fix $g::{ }^{\prime} c=>^{\prime} a$ and $w::^{\prime} c$
assume sum: $\left(\sum x \in f\right.$ ' $W$. scale $\left.C(g x) x\right)=0$ and $w: w \in f^{\prime} W$
have $0=\left(\sum x \in f\right.$ ' $W$. scale $\left.C(g x) x\right)$ using sum by simp
also have $\ldots=\operatorname{sum}((\lambda x$. scaleC $(g x) x) \circ f) W$
by (rule sum.reindex, rule inj-on-extended $[O F l$, of $C B]$ )
(unfold $W$-def, rule finite- $C$, rule independent- $C$, rule $C$-eq, simp, rule ker-in-span)
also have $\ldots=\left(\sum x \in W\right.$. scale $\left.C((g \circ f) x)(f x)\right)$ unfolding o-def .. also have $\ldots=f\left(\sum x \in W\right.$. scale $\left.((g \circ f) x) x\right)$
unfolding l.sum[symmetric] l.scale[symmetric] by simp
finally have $f$-sum-zero: $f\left(\sum x \in W\right.$. scale $\left.((g \circ f) x) x\right)=0$ by (rule sym) hence $\left(\sum x \in W\right.$. scale $\left.((g \circ f) x) x\right) \in$ ker- $f$ unfolding ker-f-def by simp hence $\exists h$. $\left(\sum v \in B\right.$. scale $\left.(h v) v\right)=\left(\sum x \in W\right.$. scale $\left.((g \circ f) x) x\right)$
using span-finite [OF finite-B] using ker-in-span
unfolding ker-f-def by force
then obtain $h$ where
sum-h: $\left(\sum v \in B\right.$. scale $\left.(h v) v\right)=\left(\sum x \in W\right.$. scale $\left.((g \circ f) x) x\right)$ by blast define $t$ where $t a=($ if $a \in B$ then $h$ a else $-((g \circ f) a))$ for $a$
have $0=\left(\sum v \in B\right.$. scale $\left.(h v) v\right)+-\left(\sum x \in W\right.$. scale $\left.((g \circ f) x) x\right)$ using sum-h by simp
also have $\ldots=\left(\sum v \in B\right.$. scale $\left.(h v) v\right)+\left(\sum x \in W .-(\right.$ scale $((g \circ f) x)$
unfolding sum-negf ..
also have $\ldots=\left(\sum v \in B\right.$. scale $\left.(t v) v\right)+\left(\sum x \in W .-(\operatorname{scale}((g \circ f) x) x)\right)$
unfolding add-right-cancel unfolding $t$-def by simp
also have $\ldots=\left(\sum v \in B\right.$. scale $\left.(t v) v\right)+\left(\sum x \in W\right.$. scale $\left.(t x) x\right)$
by (unfold add-left-cancel $t$-def $W$-def, rule sum.cong) simp+
also have $\ldots=\left(\sum v \in B \cup W\right.$. scale $\left.(t v) v\right)$
by (rule sum.union-inter-neutral [symmetric], rule finite-B, rule finite-W)
(simp add: $W$-def)
finally have $\left(\sum v \in B \cup W\right.$. scale $\left.(t v) v\right)=0$ by $\operatorname{simp}$
hence coef-zero: $\forall x \in B \cup W . t x=0$
using $C$-eq scalars-zero-if-independent [OF finite-C independent- $C$ ]
unfolding $W$-def by simp
obtain $y$ where $w$-fy: $w=f y$ and $y$-in- $W: y \in W$ using $w$ by fast
have $-g w=t y$
unfolding $t$-def $w$-fy using $y$-in- $W$ unfolding $W$-def by simp
also have $\ldots=0$ using coef-zero $y$-in- $W$ unfolding $W$-def by simp
finally show $g w=0$ by $\operatorname{simp}$
qed
qed auto
finally show card $(C-B)=l . v s 2 . \operatorname{dim}($ range $f)$ unfolding $W$-def.
qed
finally show ?thesis unfolding $V$-def ker-f-def unfolding dim-UNIV . qed
qed
end


### 5.3 The rank nullity theorem for matrices

The proof of the theorem for matrices is direct, as a consequence of the "rank nullity theorem".

```
lemma rank-nullity-theorem-matrices:
    fixes A::'a::{field}`'cols::{finite, wellorder} `'rows
    shows ncols A = vec.dim (null-space A) + vec.dim (col-space A)
    using vec.rank-nullity-theorem[OF matrix-vector-mul-linear-gen, of A]
    apply (subst (2 3) matrix-of-matrix-vector-mul [of A, symmetric])
    unfolding null-space-eq-ker[OF matrix-vector-mul-linear-gen]
    unfolding col-space-eq-range [OF matrix-vector-mul-linear-gen]
    unfolding vec.dimension-def ncols-def card-cart-basis
    by simp
end
```


## References

[1] S. Axler. Linear Algebra Done Right. Springer, 2nd edition, 1997.
[2] M. S. Gockenbach. Finite Dimensional Linear Algebra. CRC Press, 2010.


[^0]:    ${ }^{*}$ This research has been funded by the research grant FPIUR12 of the Universidad de La Rioja.

