# Rank-Nullity Theorem in Linear Algebra

By Jose Divasón and Jesús Aransay*

March 19, 2025

**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



*This research has been funded by the research grant FPIUR12 of the Universidad de La Rioja.

# 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* (λ(*x*::'*a*) *y*::'*a*. *y* ≤ *x*) (λ(*x*::'*a*) *y*::'*a*. *y* < *x*)
**shows** *class.wellorder* (λ(*x*::'*a*) *y*::'*a*. *y* ≤ *x*) (λ(*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* (≥)::('*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* (λ(*x*::'*a*) *y*::'*a*. *y* ≤ *x*) (λ(*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*: $Rep\ x \leq int\ CARD\ ('a)$
**by** (*rule Rep-less-n* [*THEN order-less-imp-le*])

**lemma** *Rep-inject-sym*: $x = y \longleftrightarrow Rep\ x = Rep\ y$
**by** (*rule type-definition.Rep-inject* [*OF type*, *symmetric*])

**lemma** *Rep-inverse*: $Abs\ (Rep\ x) = x$
**by** (*rule type-definition.Rep-inverse* [*OF type*])

**lemma** *Abs-inverse*: $m \in \{0..<int\ CARD\ ('a)\} \Longrightarrow Rep\ (Abs\ m) = m$
**by** (*rule type-definition.Abs-inverse* [*OF type*])

**lemma** *Rep-Abs-mod*: $Rep\ (Abs\ (m\ mod\ int\ CARD\ ('a))) = m\ mod\ int\ CARD\ ('a)$
**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*: $Rep\ (Abs\ 1) = 1$
**using** *size1* **by** (*simp add*: *Abs-inverse*)

**lemma** *Rep-1*: $Rep\ 1 = 1$
**by** (*simp add*: *one-def Rep-Abs-1*)

**lemma** *Rep-mod*: $Rep\ x\ mod\ int\ CARD\ ('a) = Rep\ x$
**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** $to\text{-}nat = nat \circ Rep$

**definition** *Abs'* :: $int => 'a$
**where** $Abs'\ x = Abs(x\ mod\ int\ CARD\ ('a))$

**definition** *from-nat* :: *nat* ⇒ *'a*
**where** *from-nat* = (*Abs'* ∘ *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* = {*0*..<*int CARD*(*'a*)} **using** *Typedef.type-definition.Rep-range*[*OF type*] .
**qed**

**lemma** *mono-Rep*: *mono Rep* **by** (*metis strict-mono-Rep strict-mono-mono*)

**lemma** *Rep-ge-0*: *0* ≤ *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 CARD*(*'a*)} **by** (*metis inj-on-inverseI type type-definition.Abs-inverse*)
**show** *Abs* ' {*0*..<*int CARD*(*'a*)} = (*UNIV*::*'a set*) **by** (*metis type type-definition.univ*)
**qed**

**corollary** *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 CARD*(*'a*)}
**unfolding** *inj-on-def Abs'-def*
**by** (*auto*, *metis Rep-Abs-mod mod-pos-pos-trivial*)
**show** *Abs'* ' {*0*..<*int CARD*(*'a*)} = (*UNIV*::*'a set*)
**proof** (*unfold image-def Abs'-def*, *auto*)
**fix** *x* **show** ∃*xa*∈{*0*..<*int CARD*(*'a*)}. *x* = *Abs xa*
**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*) {*0*..<*CARD*(*'a*)} (*UNIV*::*'a set*)
**proof** (*unfold bij-betw-def*, *rule conjI*)
**have** *set-eq*: {*0*::*int*..<*int CARD*(*'a*)} = *int*' {*0*..<*CARD*(*'a*)} **apply** (*auto*)
**proof** −
**fix** *x*::*int* **assume** *x1*: (*0*::*int*) ≤ *x* **and** *x2*: *x* < *int CARD*(*'a*) **show** *x* ∈ *int* ' {*0*::*nat*..<*CARD*(*'a*)}
**proof** (*unfold image-def*, *auto*, *rule bexI*[*of - nat x*])
**show** *x* = *int* (*nat x*) **using** *x1* **by** *auto*
**show** *nat x* ∈ {*0*::*nat*..<*CARD*(*'a*)} **using** *x1 x2* **by** *auto*
**qed**
**qed**
**show** *inj-on* (*from-nat*::*nat*⇒*'a*) {*0*::*nat*..<*CARD*(*'a*)}
**proof** (*unfold from-nat-def* , *rule comp-inj-on*)
**show** *inj-on int* {*0*::*nat*..<*CARD*(*'a*)} **by** (*metis inj-of-nat subset-inj-on top-greatest*)
**show** *inj-on* (*Abs'*::*int*=>*'a*) (*int* ' {*0*::*nat*..<*CARD*(*'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*=>'*a*)` {*0*::*nat*..<*CARD*('*a*)} = *UNIV*
**unfolding** *from-nat-def* **using** *bij-Abs'*
**unfolding** *bij-betw-def set-eq o-def* **by** *blast*
**qed**

**lemma** *to-nat-is-inv*: *the-inv-into* {*0*..<*CARD*('*a*)} (*from-nat*::*nat*=>'*a*) = (*to-nat*::'*a*=>*nat*)
**proof** (*unfold the-inv-into-def fun-eq-iff from-nat-def to-nat-def o-def*, *clarify*)
**fix** *x*::'*a* **show** (*THE y*::*nat*. *y* ∈ {*0*::*nat*..<*CARD*('*a*)} ∧ *Abs'* (*int y*) = *x*) = *nat* (*Rep x*)
**proof** (*rule the-equality*, *auto*)
**show** *Abs'* (*Rep x*) = *x* **by** (*metis Abs'-def Rep-inverse Rep-mod*)
**show** *nat* (*Rep x*) < *CARD*('*a*) **by** (*metis* (*full-types*) *Rep-less-n nat-int size0 zless-nat-conj*)
**assume** *x*: ¬ (*0*::*int*) ≤ *Rep x* **show** (*0*::*nat*) < *CARD*('*a*) **and** *Abs'* (*0*::*int*) = *x*
**using** *Rep-ge-0 x* **by** *auto*
**next**
**fix** *y*::*nat* **assume** *y*: *y* < *CARD*('*a*)
**have** (*Rep*(*Abs'*(*int y*)::'*a*)) = (*Rep*((*Abs*(*int y mod int CARD*('*a*)))::'*a*)) **unfolding** *Abs'-def* **..**
**also have** ... = (*Rep* (*Abs* (*int y*)::'*a*)) **using** *zmod-int*[*of y CARD*('*a*)]
**using** *y mod-less* **by** *auto*
**also have** ... = (*int y*) **proof** (*rule Abs-inverse*) **show** *int y* ∈ {*0*::*int*..<*int CARD*('*a*)}
**using** *y* **by** *auto* **qed**
**finally show** *y* = *nat* (*Rep* (*Abs'* (*int y*)::'*a*)) **by** (*metis nat-int*)
**qed**
**qed**

**lemma** *bij-to-nat*: *bij-betw* (*to-nat*) (*UNIV*::'*a set*) {*0*..<*CARD*('*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* ∈ (*UNIV*::'*a set*)) = *0*
**proof** (*rule Least-equality*, *auto*)
**fix** *y*::'*a*
**have** (*0*::'*a*) ≤ *Abs* (*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** ... = *y* **by** (*metis Rep-inverse Rep-mod*)
**finally show** (*0*::'*a*) ≤ *y* **.**

**qed**

**lemma** *add-to-nat-def*: $x + y = \textit{from-nat}\ (\textit{to-nat}\ x + \textit{to-nat}\ y)$
**unfolding** *from-nat-def to-nat-def o-def* **using** *Rep-ge-0*[*of x*] **using** *Rep-ge-0*[*of y*]
**using** *Rep-less-n*[*of x*] *Rep-less-n*[*of y*]
**unfolding** *Abs'-def* **unfolding** *add-def*[*of x y*] **by** *auto*

**lemma** *to-nat-1*: *to-nat 1 = 1*
**by** (*simp add*: *to-nat-def Rep-1*)

**lemma** *add-def'*:
**shows** $x + y = \textit{Abs'}\ (\textit{Rep}\ x + \textit{Rep}\ y)$ **unfolding** *Abs'-def* **using** *add-def* **by** *simp*

**lemma** *Abs'-0*:
**shows** $\textit{Abs'}\ (\textit{CARD}('a)) = (0::'a)$ **by** (*metis* (*opaque-lifting*, *mono-tags*) *Abs'-def mod-self zero-def*)

**lemma** *Rep-plus-one-le-card*:
**assumes** *a*: $a + 1 \neq 0$
**shows** $(\textit{Rep}\ a) + 1 < \textit{CARD}\ ('a)$
**proof** (*rule ccontr*)
**assume** $\neg\ \textit{Rep}\ a + 1 < \textit{CARD}('a)$ **hence** *to-nat-eq-card*: $\textit{Rep}\ a + 1 = \textit{CARD}('a)$
**using** *Rep-less-n*
**by** (*simp add*: *add1-zle-eq order-class.less-le*)
**have** $a+1 = \textit{Abs'}\ (\textit{Rep}\ a + \textit{Rep}\ (1::'a))$ **using** *add-def'* **by** *auto*
**also have** $\ldots = \textit{Abs'}\ ((\textit{Rep}\ a) + 1)$ **using** *Rep-1* **by** *simp*
**also have** $\ldots = \textit{Abs'}\ (\textit{CARD}('a))$ **unfolding** *to-nat-eq-card* **..**
**also have** $\ldots = 0$ **using** *Abs'-0* **by** *auto*
**finally show** *False* **using** *a* **by** *contradiction*
**qed**

**lemma** *to-nat-plus-one-less-card*: $\forall\, a.\ a+1 \neq 0 \longrightarrow \textit{to-nat}\ a + 1 < \textit{CARD}('a)$
**proof** (*clarify*)
**fix** *a*
**assume** *a*: $a + 1 \neq 0$
**have** $\textit{Rep}\ a + 1 < \textit{int}\ \textit{CARD}('a)$ **using** *Rep-plus-one-le-card*[*OF a*] **by** *auto*
**hence** $\textit{nat}\ (\textit{Rep}\ a + 1) < \textit{nat}\ (\textit{int}\ \textit{CARD}('a))$ **unfolding** *zless-nat-conj* **using** *size0* **by** *fast*
**thus** $\textit{to-nat}\ a + 1 < \textit{CARD}('a)$ **unfolding** *to-nat-def o-def* **using** *nat-add-distrib*[*OF Rep-ge-0*] **by** *simp*
**qed**

**corollary** *to-nat-plus-one-less-card'*:
**assumes** $a+1 \neq 0$
**shows** $\textit{to-nat}\ a + 1 < \textit{CARD}('a)$ **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* ⟷ *x* = *y*
**using** *injD* [*OF bij-betw-imp-inj-on*[*OF bij-to-nat*]] **by** *blast*

**lemma** *mod-type-forall-eq* [*simp*]: (∀ *j*::′*a*. (*to-nat j*)<*CARD*(′*a*) ⟶ *P j*) = (∀ *a*. *P a*)
**proof** (*auto*)
**fix** *a* **assume** *a*: ∀ *j*. (*to-nat*::′*a*=>*nat*) *j* < *CARD*(′*a*) ⟶ *P j*
**have** (*to-nat*::′*a*=>*nat*) *a* < *CARD*(′*a*) **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** ... = *from-nat* (*the-inv-into* {*0*..<*CARD*(′*a*)} (*from-nat*) *j*) **unfolding** *to-nat-is-inv* **..**
**also have** ... = *j*
**proof** (*rule f-the-inv-into-f*)
**show** *inj-on from-nat* {*0*..<*CARD*(′*a*)} **by** (*metis bij-betw-imp-inj-on bij-from-nat*)
**show** *j* ∈ *from-nat* ‘ {*0*..<*CARD*(′*a*)} **by** (*metis UNIV-I bij-betw-def bij-from-nat*)
**qed**
**finally show** *from-nat k* = *j* **.**
**qed**

**lemma** *to-nat-mono*:
**assumes** *ab*: *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** *ab*: *a* ≤ *b*
**shows** *to-nat a* ≤ *to-nat b*
**proof** (*cases a*=*b*)
**case** *True* **thus** *?thesis* **by** *auto*
**next**
**case** *False*
**hence** *a*<*b* **using** *ab* **by** *simp*
**thus** *?thesis* **using** *to-nat-mono* **by** *fastforce*
**qed**

**lemma** *least-mod-type*:
**shows** *0* ≤ (*n*::′*a*)

**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* **unfolding** *bij-betw-def* **by** *auto*
**show** *x* ∈ {*0*..<*CARD*('*a*)} **using** *x* **by** *simp*
**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 j t 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* ≤ *j* **and** *j*<*CARD* ('*a*)
**shows** (*from-nat i*::'*a*) ≤ *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* < *from-nat k*
**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* ≤ *b*
**proof** −
**have** *a* + *1* = (*from-nat* (*to-nat* (*a* + *1*))::'*a*) **using** *from-nat-to-nat-id* [*of a*+*1*,*symmetric*] **.**
**also have** ... ≤ (*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* ≤ *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* ≤ *to-nat b*› *preorder-class.le-less-trans*)
**thus** *to-nat* (*a* + *1*) ≤ *to-nat b* **by** (*metis* ‹*to-nat a* + *1* ≤ *to-nat b*› *to-nat-suc*)
**qed**
**also have** ... = *b* **by** (*metis from-nat-to-nat-id*)
**finally show** *a* + (*1*::'*a*) ≤ *b* **.**
**qed**

**lemma** *le-Suc'*:
**assumes** *ab*: *a* + *1* ≤ *b*
**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*) ≤ *to-nat b* **using** *ab* **by** (*metis to-nat-mono'*)
**hence** *to-nat* (*a*) + *1* ≤ *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 by (metis to-nat-from-nat)
qed

lemma Suc-le:
  assumes less-card: (to-nat a) + 1 < CARD ('a)
  shows a < a + 1
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 by (metis to-nat-from-nat)
qed

lemma Suc-le':
  fixes a::'a
  assumes a + 1 ≠ 0
  shows a < a + 1 using Suc-le to-nat-plus-one-less-card assms by blast

lemma from-nat-not-eq:
  assumes a-eq-to-nat: a ≠ to-nat b
  and a-less-card: a<CARD('a)
  shows from-nat a ≠ b
proof (rule ccontr)
  assume ¬ from-nat a ≠ b hence from-nat a = b 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 ≠ j
  shows i+1<j by (metis assms le-Suc le-neq-trans)


lemma Greatest-is-minus-1: ∀ a::'a. a ≤ −1
proof (clarify)
  fix a::'a
  have zero-ge-card-1: 0 ≤ int CARD('a) − 1 using size1 by auto
  have card-less: int CARD('a) − 1 < int CARD('a) by auto
  have not-zero: 1 mod int CARD('a) ≠ 0
    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** ... ≤ *Abs′* (*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*) ≤ *CARD*('*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* ≤ −*1* **by** *fastforce*
**qed**

**lemma** *a-eq-minus-1*: ∀ *a*::'*a*. *a*+*1* = *0* ⟶ *a* = −*1*
**by** (*metis eq-neg-iff-add-eq-0*)

**lemma** *forall-from-nat-rw*:
**shows** (∀ *x*∈{*0*..<*CARD*('*a*)}. *P* (*from-nat x*::'*a*)) = (∀ *x*. *P* (*from-nat x*))
**proof** (*auto*)
**fix** *y* **assume** ∗: ∀ *x*∈{*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*∈{*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 ≠ 0*
**proof** (*rule ccontr*, *simp*)
**assume** *a-plus-one-zero*: *a + 1 = 0*
**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 + 1 < 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* ∈ {*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** (*Abs*::*int* => *'a bit0*) *x = Abs-bit0' x*

**definition** (*Rep*::*'a bit1* => *int*) *x = Rep-bit1 x*
**definition** (*Abs*::*int* => *'a bit1*) *x = Abs-bit1' x*

**instance**
**proof**
**show** (*0*::*'a bit0*) = *Abs* (*0*::*int*) **unfolding** *Abs-bit0-def Abs-bit0'-def zero-bit0-def*
**by** *auto*
**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** ∀ *x*::*'a bit0*. *Rep-bit0 x* ∈ {*0*::*int*..<*int CARD*(*'a bit0*)}
**unfolding** *card-bit0* **unfolding** *of-nat-mult*
**using** *Rep-bit0* [**where** *?'a* = *'a*] **by** *simp*
**show** ∀ *x*::*'a bit0*. *Abs-bit0* (*Rep-bit0 x mod int CARD*(*'a bit0*)) = *x*
**by** (*metis Rep-bit0-inverse bit0.Rep-mod*)
**show** ∀ *y*::*int*. *y* ∈ {*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-bit0-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* ((*Rep x* + *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* ((*Rep x* − *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** ∀ *x*::*'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** ∀ *x*::*'a bit1*. *Abs-bit1* (*Rep-bit1 x mod int CARD*(*'a bit1*)) = *x*

**by** (*metis Rep-bit1-inverse bit1.Rep-mod*)
**show** $\forall$ *y*::*int*. *y* $\in$ {*0*::*int*..<*int CARD*('*a bit1*)}
$\longrightarrow$ *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*^'*columns*^'*rows* => *nat*
**where** *nrows A* = *CARD*('*rows*)

**definition** *ncols* :: '*a*^'*columns*^'*rows* => *nat*
**where** *ncols A* = *CARD*('*columns*)

**definition** *matrix-scalar-mult* :: '*a*::*ab-semigroup-mult* => '*a* ^'*n*^'*m* => '*a* ^'*n*^'*m*
(**infixl** ‹∗*k*› *70*)
**where** *k* ∗*k A* ≡ (*χ i j*. *k* ∗ *A* $ *i* $ *j*)

## 3.2 Basic properties about matrices

**lemma** *nrows-not-0*[*simp*]:
**shows** *0* ≠ *nrows A* **unfolding** *nrows-def* **by** *simp*

**lemma** *ncols-not-0*[*simp*]:
**shows** $0 \neq ncols\ A$ **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* $\lambda i.\ row\ i\ A$] **unfolding** *rows-def* **.**

**lemma** *finite-columns*: *finite* (*columns A*)
**using** *finite-Atleast-Atmost-nat*[*of* $\lambda i.\ column\ i\ A$] **unfolding** *columns-def* **.**

**lemma** *transpose-vector*: $x\ v*\ A = transpose\ A\ *v\ (x::'a::comm\text{-}semiring\text{-}1\hat{}'m)$
**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** $x :: 'a::\{field\}\hat{}'n$ **and** $A :: 'a\hat{}'m\hat{}'n$
**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** (*simp add*: *sum-distrib-left vector-space-over-itself.scale-scale*)
(*simp add*: *mult.commute*)

**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** $A :: 'a::\{field\}\hat{}'m\hat{}'n$
**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** $A :: 'a::\{field\}\hat{}'m\hat{}'n$
**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* **where**
*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* ≠ *0* **by** *simp*
**with** *card-eq-0-iff* [*of S*] **show** *finite S* **by** *simp*
**qed**

**lemma** *independent-is-basis*:
**fixes** *B* :: ('*a*::{*field*}^'*n*) *set*
**shows** *vec.independent B* ∧ *card B* = *CARD*('*n*) ⟷ *is-basis B*
**proof**
**assume** *vec.independent B* ∧ *card B* = *CARD*('*n*)
**hence** *vec.independent B* **and** *card B* = *CARD*('*n*) **by** *simp*+
**from** *card-finite* [*of B*, **where** '*n* = '*n*] **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* ⊆ *UNIV* **by** *simp*
**moreover**
**{ from** ‹*is-basis B*› **have** *UNIV* ⊆ *vec.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*) *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*] **show** *finite B* **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* ⟹ *finite ?B* ∧ *card ?B* = *local.dim* (*local.span ?B*)

**lemma** *card-eq-dim-span-indep*:
**assumes** *dim* (*span A*) = *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** *dim-A*: *dim A* = *0*
**shows** *A* = {} ∨ *A* = {*0*}
**using** *dim-A local.card-ge-dim-independent local.independent-empty* **by** *force*

**lemma** *dim-zero-eq'*:
**assumes** *A*: *A* = {} ∨ *A* = {*0*}
**shows** *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** (*dim A* = *0*) = (*A* = {*0*})
**by** (*metis dim-zero-eq dim-zero-eq' subspace-0*[*OF subs-A*] *empty-iff*)

**lemma** *span-0-imp-set-empty-or-0*:
**assumes** *span A* = {*0*}
**shows** *A* = {} ∨ *A* = {*0*} **by** (*metis assms span-superset subset-singletonD*)

**end**

**context** *Vector-Spaces.linear*
**begin**

**lemma** *linear-injective-ker-0*:

**shows** *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*::*'a*::{*comm-ring-1*}^*'n*^*'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*::*'a*::{*comm-ring-1*}^*'n*^*'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* · *column j B*
**unfolding** *inner-vec-def matrix-matrix-mult-def row-def column-def* **by** *auto*

**lemma** *matrix-vmult-column-sum*:
**fixes** *A*::*'a*::{*field*}^*'n*^*'m*
**shows** ∃*f*. *A* ∗*v x* = *sum* (λ*y*. *f y* ∗*s y*) (*columns A*)
**proof** (*rule exI*[*of* - λ*y*. *sum* (λ*i*. *x* $ *i*) {*i*. *y* = *column i A*}])
**let** *?f*=λ*y*. *sum* (λ*i*. *x* $ *i*) {*i*. *y* = *column i A*}
**let** *?g*=(λ*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*: ⋃ (*?g*‘(*columns* (*A*))) = *UNIV* **unfolding** *columns-def* **by**
*auto*
**have** *A* ∗*v x* = (∑ *i*∈*UNIV*. *x* $ *i* ∗*s column i A*) **unfolding** *matrix-mult-sum*
**..**
**also have** ... = *sum* (λ*i*. *x* $ *i* ∗*s column i A*) (⋃(*?g*‘(*columns A*))) **unfolding**
*union-univ* **..**
**also have** ... = *sum* (*sum* ((λ*i*. *x* $ *i* ∗*s column i A*))) (*?g*‘(*columns A*))
**by** (*rule sum.Union-disjoint*[*unfolded o-def*], *auto*)
**also have** ... = *sum* ((*sum* ((λ*i*. *x* $ *i* ∗*s column i A*))) ∘ *?g*) (*columns A*)
**by** (*rule sum.reindex*, *simp add*: *inj*)

**also have** ... = *sum* (λ*y*. *?f y* ∗*s y*) (*columns A*)
**proof** (*rule sum.cong*, *unfold o-def*)
**fix** *xa*
**have** *sum* (λ*i*. *x* $ *i* ∗*s column i A*) {*i*. *xa* = *column i A*}
= *sum* (λ*i*. *x* $ *i* ∗*s xa*) {*i*. *xa* = *column i A*} **by** *simp*
**also have** ... = *sum* (λ*i*. *x* $ *i*) {*i*. *xa* = *column i A*} ∗*s xa*
**using** *vec.scale-sum-left*[*of* (λ*i*. *x* $ *i*) {*i*. *xa* = *column i A*} *xa*] **..**
**finally show** (∑ *i* | *xa* = *column i A*. *x* $ *i* ∗*s column i A*) = (∑ *i* | *xa* = *column i A*. *x* $ *i*) ∗*s xa* **.**
**qed** *rule*
**finally show** *A* ∗*v x* = (∑ *y*∈*columns A*. (∑ *i* | *y* = *column i A*. *x* $ *i*) ∗*s y*) **.**
**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* λ *N*. *M* ∗∗ *N* = *mat 1* ∧ *N* ∗∗ *M* = *mat 1*]
**unfolding** *invertible-def* **and** *matrix-inv-def*
**by** *simp-all*

In the library, *matrix-inv ?A* = (*SOME A′*. *?A* ∗∗ *A′* = *mat 1* ∧ *A′* ∗∗ *?A* = *mat 1*) allows the use of non squary matrices. The following lemma can be also proved fixing *A*

**lemma** *matrix-inv-unique*:
**fixes** *A*::*′a*::{*semiring-1*}^*′n*^*′n*
**assumes** *AB*: *A* ∗∗ *B* = *mat 1* **and** *BA*: *B* ∗∗ *A* = *mat 1*
**shows** *matrix-inv A* = *B*
**by** (*metis AB 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*::*'a*::{*field*}^*'n*^*m*
**assumes** *ind-B*: *vec.independent B* **and** *inv-P*: *invertible P*
**shows** *vec.independent* (((∗*v*) *P*)‘ *B*)
**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*::*'a*::{*field*}^*'n*^*n*
**assumes** *ind-B*: *vec.independent* (((∗*v*) *P*)‘ *B*) **and** *inv-P*: *invertible P*
**shows** *vec.independent B*
**proof** −
**have** *vec.independent* (((∗*v*) (*matrix-inv P*))‘ (((∗*v*) *P*)‘ *B*))
**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*))‘ (((∗*v*) *P*)‘ *B*) = *B*
**proof** (*auto*)
**fix** *x* **assume** *x*: *x* ∈ *B* **show** *matrix-inv P* ∗*v* (*P* ∗*v x*) ∈ *B*
**by** (*metis* (*full-types*) *x inv-P matrix-inv-left matrix-vector-mul-assoc matrix-vector-mul-lid*)
**thus** *x* ∈ (∗*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*::{*field*}) *TYPE*(*'rows*::{*mod-type*})=*CARD*(*'r*
**proof** −
**let** *?f*=λ*x*. *axis* (*from-nat x*) *1*::*'a*^*'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** ... = *card*{..<*CARD*(*'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* < *CARD*(*'rows*) **and** *y*: *y* < *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* ∈ (λ*x*. *axis* (*from-nat x*::*'rows*) *1*) ‘ {..<*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* = (λ*i*. *a* ∗*R f i* )

**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* :: ('*a*, '*b*::*finite*) *vec* => ('*a*, '*b*::*finite*) *vec* => *bool*
 **where** *equal-vec x y* = (∀ *i*. *x*$*i* = *y*$*i*)
**instance**
**proof** (*intro-classes*)
 **fix** *x y*::('*a*, '*b*::*finite*) *vec*
 **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*))::'*a*::{*field*}=>'*a*^'*cols*^'*rows*=>'*a*^'*cols*^'*rows*
**proof** (*unfold-locales*)
**fix** *a*::'*a* **and** *x y*::'*a*^'*cols*^'*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*::'*a* **and** *x*::'*a*^'*cols*^'*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*
**imports**
*Miscellaneous*
**begin**

## 4.1 The fundamental subspaces of a matrix

### 4.1.1 Definitions

**definition** *left-null-space* :: *'a*::{*semiring-1*}^*'n*^*'m* => (*'a*^*'m*) *set*
**where** *left-null-space A* = {*x*. *x v*∗ *A* = *0*}

**definition** *null-space* :: *'a*::{*semiring-1*}^*'n*^*'m* => (*'a*^*'n*) *set*
**where** *null-space A* = {*x*. *A* ∗*v x* = *0*}

**definition** *row-space* :: *'a*::{*field*}^*'n*^*'m*=>(*'a*^*'n*) *set*
**where** *row-space A* = *vec.span* (*rows A*)

**definition** *col-space* :: *'a*::{*field*}^*'n*^*'m*=>(*'a*^*'m*) *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*::*'a*::{*comm-semiring-1*}^*'n*^*'m*) = *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*::*'a*::{*comm-semiring-1*}^*'n*^*'m*) = *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*::*'a*::{*field*}^*'columns*^*'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*::*'a*::{*field*}^*'n*^*'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*::*'a*::{*field*}^*'n*^*'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*^*'m*
**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*::*'a*::{*field*}^*'m*::{*finite*, *wellorder*}^*'n*
**shows** *col-space A* = {*y*. ∃ *x*. *A* ∗*v x* = *y*}
**proof** (*unfold col-space-def vec.span-finite*[*OF finite-columns*], *auto*)
**fix** *x*
**show** *A* ∗*v x* ∈ *range* (λ*u*. ∑ *v*∈*columns A*. *u v* ∗*s v*) **using** *matrix-vmult-column-sum*[*of A x*] **by** *auto*
**next**
**fix** *u*::(*'a*, *'n*) *vec* ⇒ *'a*
**let** *?g*=λ*y*. {*i*. *y*=*column i A*}
**let** *?x*=(χ *i*. *if i*=(*LEAST a*. *a* ∈ *?g* (*column i A*)) *then u* (*column i A*) *else 0*)
**show** ∃ *x*. *A* ∗*v x* = (∑ *v*∈*columns A*. *u v* ∗*s v*)
**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*: ⋃(*?g*‘(*columns A*)) = *UNIV* **unfolding** *columns-def* **by** *auto*

**have** *sum* (λ*i*.(*if i* = (*LEAST a*. *column i A* = *column a A*) *then u* (*column i A*) *else 0*) ∗*s column i A*) *UNIV*
= *sum* (λ*i*. (*if i* = (*LEAST a*. *column i A* = *column a A*) *then u* (*column i A*) *else 0*) ∗*s column i A*) (⋃(*?g*‘(*columns A*)))
**unfolding** *union-univ* **..**
**also have** ... = *sum* (*sum* (λ*i*.(*if i* = (*LEAST a*. *column i A* = *column a A*) *then u* (*column i A*) *else 0*) ∗*s column i A*)) (*?g*‘(*columns A*))
**by** (*rule sum.Union-disjoint*[*unfolded o-def*], *auto*)
**also have** ... = *sum* ((*sum* (λ*i*.(*if i* = (*LEAST a*. *column i A* = *column a A*) *then u* (*column i A*) *else 0*) ∗*s column i A*)) ∘ *?g*) (*columns A*) **by** (*rule sum.reindex*, *simp add*: *inj*)
**also have** ... = *sum* (λ*y*. *u y* ∗*s y*) (*columns A*)
**proof** (*rule sum.cong*, *auto*)
**fix** *x*
**assume** *x-in-cols*: *x* ∈ *columns A*
**obtain** *b* **where** *b*: *x*=*column b A* **using** *x-in-cols* **unfolding** *columns-def* **by** *blast*
**let** *?f*=(λ*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} - {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} - {LEAST a. x = column a A})
      apply (rule sum.remove, auto, rule LeastI-ex)
      using x-in-cols unfolding columns-def by auto
    also have ... = ?f (LEAST a. x = column a A) unfolding sum-rw by simp
    also have ... = u x *s x
    proof (auto, rule LeastI2)
      show x = column b A using b .
      fix xa
      assume x: x = column xa A
      show u (column xa A) *s column xa A = u x *s x unfolding x ..
    next
      assume (LEAST a. x = column a A) ≠ (LEAST a. column (LEAST c. x
= column c A) A = column a A)
     moreover have (LEAST a. x = column a A) = (LEAST a. column (LEAST
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 (∑ i | x = column i A. (if i = (LEAST 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 (∑ i∈UNIV. (if i = (LEAST a. column i A = column a A) then
u (column i A) else 0) *s column i A) = (∑ y∈columns A. u y *s y) .
 qed
qed


corollary col-space-eq':
 fixes A::'a::{field}^'m::{finite, wellorder}^'n
 shows col-space A = range (λx. A *v x)
 unfolding col-space-eq by auto

lemma row-space-eq:
 fixes A::'a::{field}^'m^'n::{finite, wellorder}
 shows row-space A = {w. ∃ y. (transpose A) *v y = w}
 unfolding row-space-eq-col-space-transpose col-space-eq ..

lemma null-space-eq-ker:
 fixes f::('a::field^'n) => ('a^'m)
 assumes lf: Vector-Spaces.linear (*s) (*s) f
 shows null-space (matrix f) = {x. f x = 0}
 unfolding null-space-def using matrix-works [OF lf] by auto
```

**lemma** *col-space-eq-range*:
**fixes** $f$::($'a$::*field*$^{\wedge}{}'n$::{*finite*, *wellorder*}) $\Rightarrow$ ($'a^{\wedge}{}'m$)
**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*}$^{\wedge}{}'cols^{\wedge}{}'rows$
**assumes** $P$: *invertible* $P$
**shows** *null-space* ($P$**$A$) = *null-space* $A$
**unfolding** *null-space-def*
**using** $P$ *matrix-inv-left matrix-left-invertible-ker matrix-vector-mul-assoc matrix-vector-mult-0-right*
**by** *metis*

**lemma** *row-space-is-preserved*:
**fixes** $A$::$'a$::{*field*}$^{\wedge}{}'cols^{\wedge}{}'rows$::{*finite*, *wellorder*}
**and** $P$::$'a$::{*field*}$^{\wedge}{}'rows$::{*finite*, *wellorder*}$^{\wedge}{}'rows$::{*finite*, *wellorder*}
**assumes** $P$: *invertible* $P$
**shows** *row-space* ($P$**$A$) = *row-space* $A$
**proof** (*auto*)
**fix** $w$
**assume** $w$: $w \in$ *row-space* ($P$**$A$)
**from** *this* **obtain** $y$ **where** *w-By*: $w$=(*transpose* ($P$**$A$)) $*v$ $y$
**unfolding** *row-space-eq*[*of* $P$ ** $A$ ] **by** *fast*
**have** $w$ = (*transpose* ($P$**$A$)) $*v$ $y$ **using** *w-By* **.**
**also have** ... = ((*transpose* $A$) ** (*transpose* $P$)) $*v$ $y$ **unfolding** *matrix-transpose-mul*
**..**
**also have** ... = (*transpose* $A$) $*v$ ((*transpose* $P$) $*v$ $y$) **unfolding** *matrix-vector-mul-assoc*
**..**
**finally show** $w \in$ *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* ($P$**$A$) $*v$ (*transpose* (*matrix-inv* $P$) $*v$ $y$)
**unfolding** *matrix-vector-mul-assoc* **..**
**finally show** $w \in$ *row-space* ($P$**$A$) **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*: $A \subseteq B$
  **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*: $(\sum x{\in}A.\ scale\ (f\ x)\ x) = 0$
  **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

$[\![finite\ A;\ local.independent\ A;\ (\sum x{\in}A.\ f\ x *s\ x) = 0]\!] \Longrightarrow \forall\, x{\in}A.\ f\ x = 0$

holds:

**corollary** *scalars-zero-if-independent-euclidean*:
  **assumes** *ind*: *independent A*
    **and** *sum*: $(\sum x{\in}A.\ scale\ (f\ x)\ x) = 0$
  **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 $V'$, and given a basis $B$ of *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*: $C = B \cup W$
  **and** *disj-set*: $B \cap W = \{\}$
  **and** *span-B*: $\{x.\ f\ x = 0\} \subseteq span\ B$
  **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* [*OF - 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** $x$::$'b$ **and** $y$::$'b$
  **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** $\exists g.\ (\sum v{\in}B.\ scale\ (g\ v)\ v) = (x - y)$ **using** *span-B*
    **unfolding** *span-finite* [*OF fin-B*] **by** *force*
  **then obtain** *g* **where** *sum*: $(\sum v{\in}B.\ scale\ (g\ v)\ v) = (x - y)$ **by** *blast*
  — We define one of the elements as a linear combination of the second element and the ones in *B*
  **define** $h :: 'b \Rightarrow 'a$ **where** $h\ a = (if\ a = y\ then\ 1\ else\ g\ a)$ **for** *a*
  **have** $x = y + (\sum v{\in}B.\ scale\ (g\ v)\ v)$ **using** *sum* **by** *auto*
  **also have** ... = $scale\ (h\ y)\ y\ + (\sum v{\in}B.\ scale\ (g\ v)\ v)$ **unfolding** *h-def* **by** *simp*
  **also have** ... = $scale\ (h\ y)\ y + (\sum v{\in}B.\ scale\ (h\ v)\ v)$
    **apply** (*unfold add-left-cancel*, *rule sum.cong*)
    **using** *y h-def empty-iff disj-set* **by** *auto*

**also have** ... = ($\sum$ *v*∈(*insert y B*). *scale* (*h v*) *v*)
**by** (*rule sum.insert*[*symmetric*], *rule fin-B*)
(*metis* (*lifting*) *IntI disj-set empty-iff y*)
**finally have** *x-in-span-yB*: *x* ∈ *span* (*insert y B*)
**unfolding** *span-finite*[*OF 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-yB 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* = *dim* {*x*. *f x* = *0*} + *vector-space.dim scaleC* (*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* = *UNIV*
**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* ⊆ {*x*. *f x* = *0*}
**and** *independent-B*: *independent B*
**and** *ker-in-span*:{*x*. *f x* = *0*} ⊆ *span B*
**and** *card-B*: *card B* = *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** *eq-fC*: $f\ `\ C = f\ `\ B \cup f\ `\ (C - B)$
**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*[*OF independent-C*] **.**
**have** *finite-fC*: *finite* $(f\ `\ C)$ **by** (*rule finite-imageI* [*OF 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* $(f\ `\ B)$ **by** (*rule finite-imageI*[*OF 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* $(ker\text{-}f) \leq dim\ V$
**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* $= dim\ V$ **unfolding** *V-def dim-UNIV dimension-def*
**by** (*metis basis-card-eq-dim dimension-def independent-Basis span-Basis top-greatest*)
**also have** $dim\ V = dim\ C$ **unfolding** *span-C dim-span* **..**
**also have** $... = card\ C$
**using** *basis-card-eq-dim* [*of C C*, *OF - span-superset independent-C*] **by** *simp*
**also have** $... = card\ (B \cup (C - B))$ **using** *C-eq* **by** *simp*
**also have** $... = card\ B + card\ (C - B)$
**by** (*rule card-Un-disjoint*[*OF finite-B finite-CB*], *fast*)
**also have** $... = dim\ ker\text{-}f + card\ (C - B)$ **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** $... = dim\ ker\text{-}f + l.vs2.dim\ (range\ f)$
**proof** (*unfold add-left-cancel*)
**define** $W$ **where** $W = C - B$
**have** *finite-W*: *finite W* **unfolding** *W-def* **using** *finite-CB* **.**
**have** *finite-fW*: *finite* $(f\ `\ W)$ **using** *finite-imageI*[*OF finite-W*] **.**
**have** *card* $W = card\ (f\ `\ W)$
**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** $... = l.vs2.dim\ (range\ f)$

— 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* ⊆ *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* :: *'b*
**have** *fV-span*: *f ‘ V* ⊆ *l.vs2.span* (*f ‘ C*)
**by** (*simp add*: *span-C l.span-image*)
**have** ∃ *g*. (∑ *x*∈*f‘C*. *scaleC* (*g x*) *x*) = *f v*
**using** *fV-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** *fv*: *f v* = (∑ *x*∈*f ‘ C*. *scaleC* (*g x*) *x*) **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*: (∑ *x*∈*f ‘ B*. *scaleC* (*g x*) *x*) = *0*
**using** *B-in-ker* **by** (*auto intro*!: *sum.neutral*)
**have** *zero-inter*: (∑ *x*∈(*f ‘ B* ∩ *f ‘ W*). *scaleC* (*g x*) *x*) = *0*
**using** *B-in-ker* **by** (*auto intro*!: *sum.neutral*)
**have** *f v* = (∑ *x*∈*f ‘ C*. *scaleC* (*g x*) *x*) **using** *fv* **.**
**also have** ... = (∑ *x*∈(*f ‘ B* ∪ *f ‘ W*). *scaleC* (*g x*) *x*)
**using** *eq-fC W-def* **by** *simp*
**also have** ... =
(∑ *x*∈*f ‘ B*. *scaleC* (*g x*) *x*) + (∑ *x*∈*f ‘ W*. *scaleC* (*g x*) *x*)
− (∑ *x*∈(*f ‘ B* ∩ *f ‘ W*). *scaleC* (*g x*) *x*)
**using** *sum-Un* [*OF finite-fB finite-fW*] **by** *simp*
**also have** ... = (∑ *x*∈*f ‘ W*. *scaleC* (*g x*) *x*)
**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* ∈ *range* (λ*u*. ∑ *v*∈*f ‘ W*. *scaleC* (*u v*) *v*) **by** *auto*
**qed**
— 2. The image set of $W$ is linearly independent:
**show** *l.vs2.independent* (*f ‘ W*)
**using** *finite-fW*
**proof** (*rule l.vs2.independent-if-scalars-zero*)
— Every linear combination (given by $gx$) of the elements of the image set of $W$ equal to zero, requires every coefficient to be zero:
**fix** *g* :: *'c* => *'a* **and** *w* :: *'c*
**assume** *sum*: (∑ *x*∈*f ‘ W*. *scaleC* (*g x*) *x*) = *0* **and** *w*: *w* ∈ *f ‘ W*
**have** *0* = (∑ *x*∈*f ‘ W*. *scaleC* (*g x*) *x*) **using** *sum* **by** *simp*
**also have** ... = *sum* ((λ*x*. *scaleC* (*g x*) *x*) ∘ *f*) *W*
**by** (*rule sum.reindex*, *rule inj-on-extended*[*OF l*, *of C B*])
(*unfold W-def*, *rule finite-C*, *rule independent-C*, *rule C-eq*, *simp*,
*rule ker-in-span*)
**also have** ... = (∑ *x*∈*W*. *scaleC* ((*g* ∘ *f*) *x*) (*f x*)) **unfolding** *o-def* **..**

```
        also have ... = f (∑x∈W. scale ((g ∘ f) x) x)
          unfolding l.sum[symmetric] l.scale[symmetric] by simp
        finally have f-sum-zero:f (∑x∈W. scale ((g ∘ f) x) x) = 0 by (rule sym)
        hence (∑x∈W. scale ((g ∘ f) x) x) ∈ ker-f unfolding ker-f-def by simp
        hence ∃h. (∑v∈B. scale (h v) v) = (∑x∈W. scale ((g ∘ f) x) x)
          using span-finite[OF finite-B] using ker-in-span
          unfolding ker-f-def by force
        then obtain h where
          sum-h: (∑v∈B. scale (h v) v) = (∑x∈W. scale ((g ∘ f) x) x) by blast
        define t where t a = (if a ∈ B then h a else − ((g ∘ f) a)) for a
        have 0 = (∑v∈B. scale (h v) v) + − (∑x∈W. scale ((g ∘ f) x) x)
          using sum-h by simp
        also have ... =  (∑v∈B. scale (h v) v) + (∑x∈W. − (scale ((g ∘ f) x)
x))
          unfolding sum-negf ..
        also have ... = (∑v∈B. scale (t v) v) + (∑x∈W. −(scale((g ∘ f) x) x))
          unfolding add-right-cancel unfolding t-def by simp
        also have ... =  (∑v∈B. scale (t v) v) + (∑x∈W. scale (t x) x)
          by (unfold add-left-cancel t-def W-def, rule sum.cong) simp+
        also have ... = (∑v∈B ∪ W. scale (t v) v)
        by (rule sum.union-inter-neutral [symmetric], rule finite-B, rule finite-W)

            (simp add: W-def)
        finally have (∑v∈B ∪ W. scale (t v) v) = 0 by simp
        hence coef-zero: ∀x∈B ∪ 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 ∈ 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 ... = 0 using coef-zero y-in-W unfolding W-def by simp
        finally show g w = 0 by simp
      qed
    qed auto
    finally show card (C − B) = l.vs2.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.