Gauss-Jordan algorithm and its applications

By Jose Divasón and Jesús Aransay*

March 19, 2025

Abstract

In this contribution, we present a formalization of the well-known Gauss-Jordan algorithm. It states that any matrix over a field can be transformed by means of elementary row operations to a matrix in reduced row echelon form. The formalization is based on the Rank Nullity Theorem entry of the AFP and on the HOL-Multivariate-Analysis session of Isabelle, where matrices are represented as functions over finite types. We have set up properly the code generator to make this representation executable. In order to improve the performance, a refinement to immutable arrays has been carried out. We have formalized some of the applications of the Gauss-Jordan algorithm. Thanks to this development, the following facts can be computed over matrices whose elements belong to a field:

- Ranks
- Determinants
- Inverses
- Bases and dimensions of the null space, left null space, column space and row space of a matrix
- Solutions of systems of linear equations (considering any case, including systems with one solution, multiple solutions and with no solution)

Code can be exported to both SML and Haskell. In addition, we have introduced some serializations (for instance, from *bit* in Isabelle to booleans in SML and Haskell, and from *rat* in Isabelle to the corresponding one in Haskell), that speed up the performance.

Contents

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

1 Reduced row echelon form

theory *Rref*
imports
Rank-Nullity-Theorem.Mod-Type
Rank-Nullity-Theorem.Miscellaneous
begin

1.1 Defining the concept of Reduced Row Echelon Form

1.1.1 Previous definitions and properties

This function returns True if each position lesser than k in a column contains a zero.

definition *is-zero-row-upt-k* :: *'rows* => *nat* =>*'a*::{*zero*}^*'columns*::{*mod-type*}^*'rows* => *bool*
where *is-zero-row-upt-k i k A* = (∀ *j*::*'columns*. (*to-nat j*) < *k* ⟶ *A* $ *i* $ *j* = *0*)

definition *is-zero-row* :: *'rows* =>*'a*::{*zero*}^*'columns*::{*mod-type*}^*'rows* => *bool*
where *is-zero-row i A* = *is-zero-row-upt-k i* (*ncols A*) *A*

lemma *is-zero-row-upt-ncols*:
fixes *A*::*'a*::{*zero*}^*'columns*::{*mod-type*}^*'rows*
shows *is-zero-row-upt-k i* (*ncols A*) *A* = (∀ *j*::*'columns*. *A* $ *i* $ *j* = *0*) **unfolding** *is-zero-row-def is-zero-row-upt-k-def ncols-def* **by** *auto*

corollary *is-zero-row-def'*:
fixes *A*::*'a*::{*zero*}^*'columns*::{*mod-type*}^*'rows*
shows *is-zero-row i A* = (∀ *j*::*'columns*. *A* $ *i* $ *j* = *0*) **using** *is-zero-row-upt-ncols* **unfolding** *is-zero-row-def ncols-def* **.**

lemma *is-zero-row-eq-row-zero*: *is-zero-row a A* = (*row a A* = *0*)
unfolding *is-zero-row-def' row-def*
unfolding *vec-nth-inverse*
unfolding *vec-eq-iff zero-index* **..**

lemma *not-is-zero-row-upt-suc*:
assumes ¬ *is-zero-row-upt-k i* (*Suc k*) *A*
and ∀ *i*. *A* $ *i* $ (*from-nat k*) = *0*
shows ¬ *is-zero-row-upt-k i k A*
using *assms from-nat-to-nat-id*
using *is-zero-row-upt-k-def less-SucE*
by *metis*

lemma *is-zero-row-upt-k-suc*:

assumes *is-zero-row-upt-k i k A*
and *A $ i $ (from-nat k) = 0*
shows *is-zero-row-upt-k i (Suc k) A*
using *assms* **unfolding** *is-zero-row-upt-k-def* **using** *less-SucE to-nat-from-nat*
by *metis*

lemma *is-zero-row-utp-0*:
shows *is-zero-row-upt-k m 0 A* **unfolding** *is-zero-row-upt-k-def* **by** *fast*

lemma *is-zero-row-utp-0'*:
shows $\forall$ *m. is-zero-row-upt-k m 0 A* **unfolding** *is-zero-row-upt-k-def* **by** *fast*

lemma *is-zero-row-upt-k-le*:
assumes *is-zero-row-upt-k i (Suc k) A*
shows *is-zero-row-upt-k i k A*
using *assms* **unfolding** *is-zero-row-upt-k-def* **by** *simp*

lemma *is-zero-row-imp-is-zero-row-upt*:
assumes *is-zero-row i A*
shows *is-zero-row-upt-k i k A*
using *assms* **unfolding** *is-zero-row-def is-zero-row-upt-k-def ncols-def* **by** *simp*

1.1.2 Definition of reduced row echelon form up to a column

This definition returns True if a matrix is in reduced row echelon form up to the column k (not included), otherwise False.

definition *reduced-row-echelon-form-upt-k* :: *'a*::{*zero*, *one*}^'*m*::{*mod-type*}^'*n*::{*finite*, *ord*, *plus*, *one*} => *nat* => *bool*
where *reduced-row-echelon-form-upt-k A k* =
(
($\forall$ *i. is-zero-row-upt-k i k A* $\longrightarrow$ $\neg$ ($\exists$ *j. j>i* $\wedge$ $\neg$ *is-zero-row-upt-k j k A*)) $\wedge$
($\forall$ *i.* $\neg$ (*is-zero-row-upt-k i k A*) $\longrightarrow$ *A $ i $ (LEAST k. A $ i $ k* $\neq$ *0) = 1*) $\wedge$
($\forall$ *i. i<i+1* $\wedge$ $\neg$ (*is-zero-row-upt-k i k A*) $\wedge$ $\neg$ (*is-zero-row-upt-k (i+1) k A*)
$\longrightarrow$ ((*LEAST n. A $ i $ n* $\neq$ *0*) < (*LEAST n. A $ (i+1) $ n* $\neq$ *0*))) $\wedge$
($\forall$ *i.* $\neg$ (*is-zero-row-upt-k i k A*) $\longrightarrow$ ($\forall$ *j. i* $\neq$ *j* $\longrightarrow$ *A $ j $ (LEAST n. A $ i $*
n $\neq$ *0) = 0*))
)

lemma *rref-upt-0*: *reduced-row-echelon-form-upt-k A 0*
unfolding *reduced-row-echelon-form-upt-k-def is-zero-row-upt-k-def* **by** *auto*

lemma *rref-upt-condition1*:
assumes *r*: *reduced-row-echelon-form-upt-k A k*
shows ($\forall$ *i. is-zero-row-upt-k i k A* $\longrightarrow$ $\neg$ ($\exists$ *j. j>i* $\wedge$ $\neg$ *is-zero-row-upt-k j k A*))
using *r* **unfolding** *reduced-row-echelon-form-upt-k-def* **by** *simp*

lemma *rref-upt-condition2*:
assumes *r*: *reduced-row-echelon-form-upt-k A k*

shows $(\forall i.\ \neg\ (\textit{is-zero-row-upt-k}\ i\ k\ A) \longrightarrow A\ \$\ i\ \$\ (\textit{LEAST}\ k.\ A\ \$\ i\ \$\ k \neq 0) = 1)$
using *r* **unfolding** *reduced-row-echelon-form-upt-k-def* **by** *simp*

lemma *rref-upt-condition3*:
assumes *r*: *reduced-row-echelon-form-upt-k A k*
shows $(\forall i.\ i<i+1 \wedge \neg\ (\textit{is-zero-row-upt-k}\ i\ k\ A) \wedge \neg\ (\textit{is-zero-row-upt-k}\ (i+1)\ k\ A) \longrightarrow ((\textit{LEAST}\ n.\ A\ \$\ i\ \$\ n \neq 0) < (\textit{LEAST}\ n.\ A\ \$\ (i+1)\ \$\ n \neq 0)))$
using *r* **unfolding** *reduced-row-echelon-form-upt-k-def* **by** *simp*

lemma *rref-upt-condition4*:
assumes *r*: *reduced-row-echelon-form-upt-k A k*
shows $(\forall i.\ \neg\ (\textit{is-zero-row-upt-k}\ i\ k\ A) \longrightarrow (\forall j.\ i \neq j \longrightarrow A\ \$\ j\ \$\ (\textit{LEAST}\ n.\ A\ \$\ i\ \$\ n \neq 0) = 0))$
using *r* **unfolding** *reduced-row-echelon-form-upt-k-def* **by** *simp*

Explicit lemmas for each condition

lemma *rref-upt-condition1-explicit*:
assumes *reduced-row-echelon-form-upt-k A k*
and *is-zero-row-upt-k i k A*
and *j>i*
shows *is-zero-row-upt-k j k A*
using *assms rref-upt-condition1* **by** *blast*

lemma *rref-upt-condition2-explicit*:
assumes *rref-A*: *reduced-row-echelon-form-upt-k A k*
and $\neg$ *is-zero-row-upt-k i k A*
shows $A\ \$\ i\ \$\ (\textit{LEAST}\ k.\ A\ \$\ i\ \$\ k \neq 0) = 1$
using *rref-upt-condition2 assms* **by** *blast*

lemma *rref-upt-condition3-explicit*:
assumes *reduced-row-echelon-form-upt-k A k*
and $i < i + 1$
and $\neg$ *is-zero-row-upt-k i k A*
and $\neg$ *is-zero-row-upt-k* $(i + 1)$ *k A*
shows $(\textit{LEAST}\ n.\ A\ \$\ i\ \$\ n \neq 0) < (\textit{LEAST}\ n.\ A\ \$\ (i + 1)\ \$\ n \neq 0)$
using *assms rref-upt-condition3* **by** *blast*

lemma *rref-upt-condition4-explicit*:
assumes *reduced-row-echelon-form-upt-k A k*
and $\neg$ *is-zero-row-upt-k i k A*
and $i \neq j$
shows $A\ \$\ j\ \$\ (\textit{LEAST}\ n.\ A\ \$\ i\ \$\ n \neq 0) = 0$
using *assms rref-upt-condition4* **by** *auto*

Intro lemma and general properties

lemma *reduced-row-echelon-form-upt-k-intro*:
assumes $(\forall i.\ \textit{is-zero-row-upt-k}\ i\ k\ A \longrightarrow \neg\ (\exists j.\ j>i \wedge \neg\ \textit{is-zero-row-upt-k}\ j\ k\ A))$

and ($\forall i.\ \neg$ (*is-zero-row-upt-k i k A*) $\longrightarrow$ *A $ i $* (*LEAST k. A $ i $ k* $\neq$ *0*) = *1*)
and ($\forall i.$ *i*<*i+1* $\wedge\ \neg$ (*is-zero-row-upt-k i k A*) $\wedge\ \neg$ (*is-zero-row-upt-k* (*i+1*) *k A*) $\longrightarrow$ ((*LEAST n. A $ i $ n* $\neq$ *0*) < (*LEAST n. A $* (*i+1*) *$ n* $\neq$ *0*)))
and ($\forall i.\ \neg$ (*is-zero-row-upt-k i k A*) $\longrightarrow$ ($\forall j.\ i \neq j \longrightarrow$ *A $ j $* (*LEAST n. A $ i $ n* $\neq$ *0*) = *0*))
shows *reduced-row-echelon-form-upt-k A k*
unfolding *reduced-row-echelon-form-upt-k-def* **using** *assms* **by** *fast*

lemma *rref-suc-imp-rref*:
fixes *A*::*'a*::{*semiring-1*}^*'n*::{*mod-type*}^*'m*::{*mod-type*}
assumes *r*: *reduced-row-echelon-form-upt-k A* (*Suc k*)
and *k-le-card*: *Suc k* < *ncols A*
shows *reduced-row-echelon-form-upt-k A k*
proof (*rule reduced-row-echelon-form-upt-k-intro*)
show $\forall i.\ \neg$ *is-zero-row-upt-k i k A* $\longrightarrow$ *A $ i $* (*LEAST k. A $ i $ k* $\neq$ *0*) = *1*
using *rref-upt-condition2*[*OF r*] *less-SucI* **unfolding** *is-zero-row-upt-k-def* **by** *blast*
show $\forall i.$ *i* < *i* + *1* $\wedge\ \neg$ *is-zero-row-upt-k i k A* $\wedge\ \neg$ *is-zero-row-upt-k* (*i* + *1*) *k A* $\longrightarrow$ (*LEAST n. A $ i $ n* $\neq$ *0*) < (*LEAST n. A $* (*i* + *1*) *$ n* $\neq$ *0*)
using *rref-upt-condition3*[*OF r*] *less-SucI* **unfolding** *is-zero-row-upt-k-def* **by** *blast*
show $\forall i.\ \neg$ *is-zero-row-upt-k i k A* $\longrightarrow$ ($\forall j.\ i \neq j \longrightarrow$ *A $ j $* (*LEAST n. A $ i $ n* $\neq$ *0*) = *0*)
using *rref-upt-condition4*[*OF r*] *less-SucI* **unfolding** *is-zero-row-upt-k-def* **by** *blast*
show $\forall i.$ *is-zero-row-upt-k i k A* $\longrightarrow\ \neg$ ($\exists j$>*i*. $\neg$ *is-zero-row-upt-k j k A*)
proof (*clarify*, *rule ccontr*)
fix *i j*
assume *zero-i*: *is-zero-row-upt-k i k A*
and *i-less-j*: *i* < *j*
and *not-zero-j*: $\neg$ *is-zero-row-upt-k j k A*
have *not-zero-j-suc*: $\neg$ *is-zero-row-upt-k j* (*Suc k*) *A*
using *not-zero-j* **unfolding** *is-zero-row-upt-k-def* **by** *fastforce*
hence *not-zero-i-suc*: $\neg$ *is-zero-row-upt-k i* (*Suc k*) *A*
using *rref-upt-condition1*[*OF r*] *i-less-j* **by** *fast*
have *not-zero-i-plus-suc*: $\neg$ *is-zero-row-upt-k* (*i+1*) (*Suc k*) *A*
proof (*cases j=i+1*)
case *True* **thus** *?thesis* **using** *not-zero-j-suc* **by** *simp*
next
case *False*
have *i+1*<*j* **by** (*rule Suc-less*[*OF i-less-j False*[*symmetric*]])
thus *?thesis* **using** *rref-upt-condition1*[*OF r*] *not-zero-j-suc* **by** *blast*
qed
from *this* **obtain** *n* **where** *a*: *A $* (*i+1*) *$ n* $\neq$ *0* **and** *n-less-suc*: *to-nat n* < *Suc k*
unfolding *is-zero-row-upt-k-def* **by** *blast*
have (*LEAST n. A $* (*i+1*) *$ n* $\neq$ *0*) $\leq$ *n* **by** (*rule Least-le*, *simp add*: *a*)
also have ... $\leq$ *from-nat k* **by** (*metis Suc-lessD from-nat-mono' from-nat-to-nat-id*

k-le-card less-Suc-eq-le n-less-suc ncols-def)
finally have *least-le*: (*LEAST n. A $ (i + 1) $ n ≠ 0*) ≤ *from-nat k* **.**
have *least-eq-k*: (*LEAST n. A $ i $ n ≠ 0*) = *from-nat k*
proof (*rule Least-equality*)
show *A $ i $ from-nat k ≠ 0* **using** *not-zero-i-suc zero-i* **unfolding** *is-zero-row-upt-k-def* **by** (*metis from-nat-to-nat-id less-SucE*)
show ⋀*y. A $ i $ y ≠ 0 ⟹ from-nat k ≤ y* **by** (*metis is-zero-row-upt-k-def not-le-imp-less to-nat-le zero-i*)
qed
have *i-less*: *i<i+1*
proof (*rule Suc-le', rule ccontr*)
assume ¬ *i + 1 ≠ 0* **hence** *i+1=0* **by** *simp*
hence *i=−1* **using** *diff-0 diff-add-cancel diff-minus-eq-add* **by** *metis*
hence *j <= i* **using** *Greatest-is-minus-1* **by** *blast*
thus *False* **using** *i-less-j* **by** *fastforce*
qed
have *from-nat k < (LEAST n. A $ (i+1) $ n ≠ 0)*
using *rref-upt-condition3*[*OF r*] *i-less not-zero-i-suc not-zero-i-plus-suc least-eq-k* **by** *fastforce*
thus *False* **using** *least-le* **by** *simp*
qed
qed

lemma *reduced-row-echelon-if-all-zero*:
assumes *all-zero*: ∀ *n. is-zero-row-upt-k n k A*
shows *reduced-row-echelon-form-upt-k A k*
using *assms* **unfolding** *reduced-row-echelon-form-upt-k-def is-zero-row-upt-k-def* **by** *auto*

1.1.3 The definition of reduced row echelon form

Definition of reduced row echelon form, based on *reduced-row-echelon-form-upt-k-def*

definition *reduced-row-echelon-form* :: *'a*::{*zero, one*}^*'m*::{*mod-type*}^*'n*::{*finite, ord, plus, one*} => *bool*
where *reduced-row-echelon-form A = reduced-row-echelon-form-upt-k A* (*ncols A*)

Equivalence between our definition of reduced row echelon form and the one presented in Steven Roman's book: Advanced Linear Algebra.

lemma *reduced-row-echelon-form-def'*:
reduced-row-echelon-form A =
(
(∀ *i. is-zero-row i A* ⟶ ¬ (∃ *j. j>i* ∧ ¬ *is-zero-row j A*)) ∧
(∀ *i.* ¬ (*is-zero-row i A*) ⟶ *A $ i $* (*LEAST k. A $ i $ k ≠ 0*) = *1*) ∧
(∀ *i. i<i+1* ∧ ¬ (*is-zero-row i A*) ∧ ¬ (*is-zero-row* (*i+1*) *A*) ⟶ ((*LEAST k. A $ i $ k ≠ 0*) < (*LEAST k. A $* (*i+1*) *$ k ≠ 0*))) ∧
(∀ *i.* ¬ (*is-zero-row i A*) ⟶ (∀ *j. i ≠ j* ⟶ *A $ j $* (*LEAST k. A $ i $ k ≠ 0*) = *0*))

) **unfolding** *reduced-row-echelon-form-def reduced-row-echelon-form-upt-k-def is-zero-row-def*
..

1.2 Properties of the reduced row echelon form of a matrix

lemma *rref-condition1*:
assumes *r*: *reduced-row-echelon-form A*
shows $(\forall i.\ is\text{-}zero\text{-}row\ i\ A \longrightarrow \neg\ (\exists j.\ j{>}i \wedge \neg\ is\text{-}zero\text{-}row\ j\ A))$ **using** *r* **unfolding** *reduced-row-echelon-form-def'* **by** *simp*

lemma *rref-condition2*:
assumes *r*: *reduced-row-echelon-form A*
shows $(\forall i.\ \neg\ (is\text{-}zero\text{-}row\ i\ A) \longrightarrow A\ \$\ i\ \$\ (LEAST\ k.\ A\ \$\ i\ \$\ k \neq 0) = 1)$
using *r* **unfolding** *reduced-row-echelon-form-def'* **by** *simp*

lemma *rref-condition3*:
assumes *r*: *reduced-row-echelon-form A*
shows $(\forall i.\ i{<}i{+}1 \wedge \neg\ (is\text{-}zero\text{-}row\ i\ A) \wedge \neg\ (is\text{-}zero\text{-}row\ (i{+}1)\ A) \longrightarrow ((LEAST\ n.\ A\ \$\ i\ \$\ n \neq 0) < (LEAST\ n.\ A\ \$\ (i{+}1)\ \$\ n \neq 0)))$
using *r* **unfolding** *reduced-row-echelon-form-def'* **by** *simp*

lemma *rref-condition4*:
assumes *r*: *reduced-row-echelon-form A*
shows $(\forall i.\ \neg\ (is\text{-}zero\text{-}row\ i\ A) \longrightarrow (\forall j.\ i \neq j \longrightarrow A\ \$\ j\ \$\ (LEAST\ n.\ A\ \$\ i\ \$\ n \neq 0) = 0))$
using *r* **unfolding** *reduced-row-echelon-form-def'* **by** *simp*

Explicit lemmas for each condition

lemma *rref-condition1-explicit*:
assumes *rref-A*: *reduced-row-echelon-form A*
and *is-zero-row i A*
shows $\forall j{>}i.$ *is-zero-row j A*
using *rref-condition1 assms* **by** *blast*

lemma *rref-condition2-explicit*:
assumes *rref-A*: *reduced-row-echelon-form A*
and $\neg$ *is-zero-row i A*
shows $A\ \$\ i\ \$\ (LEAST\ k.\ A\ \$\ i\ \$\ k \neq 0) = 1$
using *rref-condition2 assms* **by** *blast*

lemma *rref-condition3-explicit*:
assumes *rref-A*: *reduced-row-echelon-form A*
and *i-le*: $i < i + 1$
and $\neg$ *is-zero-row i A* **and** $\neg$ *is-zero-row* $(i + 1)$ *A*
shows $(LEAST\ n.\ A\ \$\ i\ \$\ n \neq 0) < (LEAST\ n.\ A\ \$\ (i + 1)\ \$\ n \neq 0)$
using *rref-condition3 assms* **by** *blast*

lemma *rref-condition4-explicit*:
assumes *rref-A*: *reduced-row-echelon-form A*

and ¬ *is-zero-row i A*
and $i \neq j$
shows *A $ j $ (LEAST n. A $ i $ n ≠ 0) = 0*
using *rref-condition4 assms* **by** *blast*

Other properties and equivalences

lemma *rref-condition3-equiv1*:
fixes *A*::*'a*::{*one*, *zero*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *rref*: *reduced-row-echelon-form A*
and *i-less-j*: *i*<*j*
and *j-less-nrows*: *j*<*nrows A*
and *not-zero-i*: ¬ *is-zero-row* (*from-nat i*) *A*
and *not-zero-j*: ¬ *is-zero-row* (*from-nat j*) *A*
shows (*LEAST n. A $ (from-nat i) $ n ≠ 0*) < (*LEAST n. A $ (from-nat j) $ n ≠ 0*)
using *i-less-j not-zero-j j-less-nrows*
proof (*induct j*)
case *0*
show *?case* **using** *0.prems*(*1*) **by** *simp*
next
fix *j*
assume *hyp*: *i* < *j* ⟹ ¬ *is-zero-row* (*from-nat j*) *A* ⟹ *j* < *nrows A* ⟹ (*LEAST n. A $ from-nat i $ n ≠ 0*) < (*LEAST n. A $ from-nat j $ n ≠ 0*)
and *i-less-suc-j*: *i* < *Suc j*
and *not-zero-suc-j*: ¬ *is-zero-row* (*from-nat* (*Suc j*)) *A*
and *Suc-j-less-nrows*: *Suc j* < *nrows A*
have *j-less*: (*from-nat j*::*'rows*) < *from-nat* (*j*+*1*) **by** (*rule from-nat-mono*, *auto simp add*: *Suc-j-less-nrows*[*unfolded nrows-def*])
hence *not-zero-j*: ¬ *is-zero-row* (*from-nat j*) *A* **using** *rref-condition1*[*OF rref*] *not-zero-suc-j* **unfolding** *Suc-eq-plus1* **by** *blast*
show (*LEAST n. A $ from-nat i $ n ≠ 0*) < (*LEAST n. A $ from-nat* (*Suc j*) $ *n ≠ 0*)
proof (*cases i*=*j*)
case *True*
show *?thesis*
proof (*unfold True Suc-eq-plus1 from-nat-suc*, *rule rref-condition3-explicit*)
show *reduced-row-echelon-form A* **using** *rref* .
show (*from-nat j*::*'rows*) < *from-nat j* + *1* **using** *j-less* **unfolding** *from-nat-suc* .
show ¬ *is-zero-row* (*from-nat j*) *A* **using** *not-zero-j* .
show ¬ *is-zero-row* (*from-nat j* + *1*) *A* **using** *not-zero-suc-j* **unfolding** *Suc-eq-plus1 from-nat-suc* .
qed
next
case *False*
have (*LEAST n. A $ from-nat i $ n ≠ 0*) < (*LEAST n. A $ from-nat j $ n ≠ 0*)
proof (*rule hyp*)
show *i* < *j* **using** *False i-less-suc-j* **by** *simp*
show ¬ *is-zero-row* (*from-nat j*) *A* **using** *not-zero-j* .

show $j < nrows\ A$ **using** *Suc-j-less-nrows* **by** *simp*
qed
also have ... $< (LEAST\ n.\ A\ \$\ from\text{-}nat\ (j+1)\ \$\ n \neq 0)$
proof (*unfold from-nat-suc*, *rule rref-condition3-explicit*)
show *reduced-row-echelon-form A* **using** *rref* **.**
show (*from-nat j*::*'rows*) < *from-nat j + 1* **using** *j-less* **unfolding** *from-nat-suc*
.
show ¬ *is-zero-row* (*from-nat j*) *A* **using** *not-zero-j* **.**
show ¬ *is-zero-row* (*from-nat j + 1*) *A* **using** *not-zero-suc-j* **unfolding** *Suc-eq-plus1 from-nat-suc* **.**
qed
finally show $(LEAST\ n.\ A\ \$\ from\text{-}nat\ i\ \$\ n \neq 0) < (LEAST\ n.\ A\ \$\ from\text{-}nat\ (Suc\ j)\ \$\ n \neq 0)$ **unfolding** *Suc-eq-plus1* **.**
qed
qed

corollary *rref-condition3-equiv*:
fixes *A*::*'a*::{*one*, *zero*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *rref*: *reduced-row-echelon-form A*
and *i-less-j*: *i*<*j*
and *i*: ¬ *is-zero-row i A*
and *j*: ¬ *is-zero-row j A*
shows $(LEAST\ n.\ A\ \$\ i\ \$\ n \neq 0) < (LEAST\ n.\ A\ \$\ j\ \$\ n \neq 0)$
proof (*rule rref-condition3-equiv1*[*of A to-nat i to-nat j*, *unfolded from-nat-to-nat-id*])
show *reduced-row-echelon-form A* **using** *rref* **.**
show *to-nat i* < *to-nat j* **by** (*rule to-nat-mono*[*OF i-less-j*])
show *to-nat j* < *nrows A* **using** *to-nat-less-card* **unfolding** *nrows-def* **.**
show ¬ *is-zero-row i A* **using** *i* **.**
show ¬ *is-zero-row j A* **using** *j* **.**
qed

lemma *rref-implies-rref-upt*:
fixes *A*::*'a*::{*one*,*zero*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *rref*: *reduced-row-echelon-form A*
shows *reduced-row-echelon-form-upt-k A k*
proof (*rule reduced-row-echelon-form-upt-k-intro*)
show $\forall i.\ \neg$ *is-zero-row-upt-k i k A* $\longrightarrow A\ \$\ i\ \$\ (LEAST\ k.\ A\ \$\ i\ \$\ k \neq 0) = 1$
using *rref-condition2*[*OF rref*] *is-zero-row-imp-is-zero-row-upt* **by** *blast*
show $\forall i.\ i < i + 1 \wedge \neg$ *is-zero-row-upt-k i k A* $\wedge \neg$ *is-zero-row-upt-k* $(i + 1)$ *k A* $\longrightarrow (LEAST\ n.\ A\ \$\ i\ \$\ n \neq 0) < (LEAST\ n.\ A\ \$\ (i + 1)\ \$\ n \neq 0)$
using *rref-condition3*[*OF rref*] *is-zero-row-imp-is-zero-row-upt* **by** *blast*
show $\forall i.\ \neg$ *is-zero-row-upt-k i k A* $\longrightarrow (\forall j.\ i \neq j \longrightarrow A\ \$\ j\ \$\ (LEAST\ n.\ A\ \$\ i\ \$\ n \neq 0) = 0)$
using *rref-condition4*[*OF rref*] *is-zero-row-imp-is-zero-row-upt* **by** *blast*
show $\forall i.$ *is-zero-row-upt-k i k A* $\longrightarrow \neg\ (\exists j{>}i.\ \neg$ *is-zero-row-upt-k j k A*)
proof (*auto*, *rule ccontr*)
fix *i j* **assume** *zero-i-k*: *is-zero-row-upt-k i k A* **and** *i-less-j*: *i* < *j*
and *not-zero-j-k*:¬ *is-zero-row-upt-k j k A*

have *not-zero-j*: ¬ *is-zero-row j A* **using** *is-zero-row-imp-is-zero-row-upt not-zero-j-k*
by *blast*
hence *not-zero-i*: ¬ *is-zero-row i A* **using** *rref-condition1*[*OF rref*] *i-less-j* **by** *blast*
have *Least-less*: (*LEAST n. A $ i $ n ≠ 0*) < (*LEAST n. A $ j $ n ≠ 0*) **by** (*rule*
rref-condition3-equiv[*OF rref i-less-j not-zero-i not-zero-j*])
moreover have (*LEAST n. A $ j $ n ≠ 0*) < (*LEAST n. A $ i $ n ≠ 0*)
proof (*rule LeastI2-ex*)
show ∃ *a. A $ i $ a ≠ 0* **using** *not-zero-i* **unfolding** *is-zero-row-def is-zero-row-upt-k-def*
by *fast*
fix *x* **assume** *Aix-not-0*: *A $ i $ x ≠ 0*
have *k-less-x*: *k* ≤ *to-nat x* **using** *zero-i-k Aix-not-0* **unfolding** *is-zero-row-upt-k-def*
by *force*
hence *k-less-ncols*: *k* < *ncols A* **unfolding** *ncols-def* **using** *to-nat-less-card*[*of*
x] **by** *simp*
obtain *s* **where** *Ajs-not-zero*: *A $ j $ s ≠ 0* **and** *s-less-k*: *to-nat s* < *k* **using**
not-zero-j-k **unfolding** *is-zero-row-upt-k-def* **by** *blast*
have (*LEAST n. A $ j $ n ≠ 0*) ≤ *s* **using** *Ajs-not-zero Least-le* **by** *fast*
also have ... = *from-nat* (*to-nat s*) **unfolding** *from-nat-to-nat-id* **..**
also have ... < *from-nat k* **by** (*rule from-nat-mono*[*OF s-less-k k-less-ncols*[*unfolded*
ncols-def]])
also have ... ≤ *x* **using** *k-less-x leD not-le-imp-less to-nat-le* **by** *fast*
finally show (*LEAST n. A $ j $ n ≠ 0*) < *x* **.**
qed
ultimately show *False* **by** *fastforce*
qed
qed

lemma *rref-first-position-zero-imp-column-0*:
fixes *A*::*'a*::{*one*,*zero*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *rref*: *reduced-row-echelon-form A*
and *A-00*: *A $ 0 $ 0 = 0*
shows *column 0 A = 0*
proof (*unfold column-def*, *vector*, *clarify*)
fix *i*
have *is-zero-row-upt-k 0 1 A* **unfolding** *is-zero-row-upt-k-def* **using** *A-00* **by**
(*metis One-nat-def less-Suc0 to-nat-eq-0*)
hence *is-zero-row-upt-k i 1 A* **using** *rref-upt-condition1*[*OF rref-implies-rref-upt*[*OF*
rref]] **using** *least-mod-type less-le* **by** *metis*
thus *A $ i $ 0 = 0* **unfolding** *is-zero-row-upt-k-def* **using** *to-nat-eq-0* **by** *blast*
qed

lemma *rref-first-element*:
fixes *A*::*'a*::{*one*,*zero*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *rref*: *reduced-row-echelon-form A*
and *column-not-0*: *column 0 A ≠ 0*
shows *A $ 0 $ 0 = 1*
proof (*rule ccontr*)

have *A-00-not-0*: *A $ 0 $ 0 ≠ 0*
proof (*rule ccontr*, *simp*)
assume *A-00*: *A $ 0 $ 0 = 0*
from *this* **obtain** *i* **where** *Ai0*: *A $ i $ 0 ≠ 0* **and** *i*: *i>0* **using** *column-not-0*
unfolding *column-def* **by** (*metis column-def rref rref-first-position-zero-imp-column-0*)
have *is-zero-row-upt-k 0 1 A* **unfolding** *is-zero-row-upt-k-def* **using** *A-00* **by** (*metis One-nat-def less-Suc0 to-nat-eq-0*)
moreover have ¬ *is-zero-row-upt-k i 1 A* **using** *Ai0* **by** (*metis is-zero-row-upt-k-def to-nat-0 zero-less-one*)
ultimately have ¬ *reduced-row-echelon-form A* **by** (*metis A-00 column-not-0 rref-first-position-zero-imp-column-0*)
thus *False* **using** *rref* **by** *contradiction*
qed
assume *A-00-not-1*: *A $ 0 $ 0 ≠ 1*
have *Least-eq-0*: (*LEAST n. A $ 0 $ n ≠ 0*) = *0*
proof (*rule Least-equality*)
show *A $ 0 $ 0 ≠ 0* **by** (*rule A-00-not-0*)
show ⋀*y. A $ 0 $ y ≠ 0 ⟹ 0 ≤ y* **using** *least-mod-type* **.**
qed
moreover have ¬ *is-zero-row 0 A* **unfolding** *is-zero-row-def is-zero-row-upt-k-def ncols-def* **using** *A-00-not-0* **by** *auto*
ultimately have *A $ 0 $* (*LEAST n. A $ 0 $ n ≠ 0*) = *1* **using** *rref-condition2*[*OF rref*] **by** *fast*
thus *False* **unfolding** *Least-eq-0* **using** *A-00-not-1* **by** *contradiction*
qed

end

2 Code set

theory *Code-Set*
imports
HOL−Library.Code-Cardinality
begin

The following setup could help to get code generation for List.coset, but it neither works correctly it complains that code equations for remove are missed, even when List.coset should be rewritten to an enum:

declare *minus-coset-filter* [*code del*]
declare *remove-code*(*2*) [*code del*]
declare *insert-code*(*2*) [*code del*]
declare *inter-coset-fold* [*code del*]
declare *compl-coset*[*code del*]
declare *Code-Cardinality.card'-code*(*2*)[*code del*]

code-datatype *set*

The following code equation could be useful to avoid the problem of code generation for List.coset []:

lemma [*code*]: *List.coset* (*l*::'*a*::*enum list*) = *set* (*enum-class.enum*) − *set l*
by (*metis Compl-eq-Diff-UNIV coset-def enum-UNIV*)

Now the following examples work:

value *UNIV*::*bool set*
value *List.coset* ([]::*bool list*)
value *UNIV*::*Enum.finite-2 set*
value *List.coset* ([]::*Enum.finite-2 list*)
value *List.coset* ([]::*Enum.finite-5 list*)

end

3 Code generation for vectors and matrices

theory *Code-Matrix*
imports
Rank-Nullity-Theorem.Miscellaneous
Code-Set
begin

In this file the code generator is set up properly to allow the execution of matrices represented as funcions over finite types.

lemmas *vec.vec-nth-inverse*[*code abstype*]

lemma [*code abstract*]: *vec-nth 0* = (%*x*. *0*) **by** (*metis zero-index*)
lemma [*code abstract*]: *vec-nth 1* = (%*x*. *1*) **by** (*metis one-index*)
lemma [*code abstract*]: *vec-nth* (*a* + *b*) = (%*i*. *a*$*i* + *b*$*i*) **by** (*metis vector-add-component*)
lemma [*code abstract*]: *vec-nth* (*a* − *b*) = (%*i*. *a*$*i* − *b*$*i*) **by** (*metis vector-minus-component*)
lemma [*code abstract*]: *vec-nth* (*vec n*) = (λ*i*. *n*) **unfolding** *vec-def* **by** *fastforce*
lemma [*code abstract*]: *vec-nth* (*a* ∗ *b*) = (%*i*. *a*$*i* ∗ *b*$*i*) **unfolding** *vector-mult-component*
by *auto*
lemma [*code abstract*]: *vec-nth* (*c* ∗*s x*) = (λ*i*. *c* ∗ (*x*$*i*)) **unfolding** *vector-scalar-mult-def*
by *auto*
lemma [*code abstract*]: *vec-nth* (*a* − *b*) = (%*i*. *a*$*i* − *b*$*i*) **by** (*metis vector-minus-component*)

definition *mat-mult-row*
where *mat-mult-row m m*′ *f* = *vec-lambda* (%*c*. *sum* (%*k*. ((*m*$*f*)$*k*) ∗ ((*m*′$*k*)$*c*))
(*UNIV* :: '*n*::*finite set*))

lemma *mat-mult-row-code* [*code abstract*]:
vec-nth (*mat-mult-row m m*′ *f*) = (%*c*. *sum* (%*k*. ((*m*$*f*)$*k*) ∗ ((*m*′$*k*)$*c*)) (*UNIV*
:: '*n*::*finite set*))
by(*simp add*: *mat-mult-row-def fun-eq-iff*)

lemma *mat-mult* [*code abstract*]: *vec-nth* (*m* ∗∗ *m*′) = *mat-mult-row m m*′
unfolding *matrix-matrix-mult-def mat-mult-row-def*[*abs-def*]
using *vec-lambda-beta* **by** *auto*

lemma *matrix-vector-mult-code* [*code abstract*]:
vec-nth (*A* ∗*v x*) = (%*i*. (∑ *j*∈*UNIV*. *A* $ *i* $ *j* ∗ *x* $ *j*)) **unfolding** *matrix-vector-mult-def* **by** *fastforce*

lemma *vector-matrix-mult-code* [*code abstract*]:
vec-nth (*x v*∗ *A*) = (%*j*. (∑ *i*∈*UNIV*. *x* $ *i* ∗ *A* $ *i* $ *j*)) **unfolding** *vector-matrix-mult-def* **by** *fastforce*

definition *mat-row*
where *mat-row k i* = *vec-lambda* (%*j*. *if i* = *j then k else 0*)

lemma *mat-row-code* [*code abstract*]:
vec-nth (*mat-row k i*) = (%*j*. *if i* = *j then k else 0*) **unfolding** *mat-row-def* **by** *auto*

lemma [*code abstract*]: *vec-nth* (*mat k*) = *mat-row k*
unfolding *mat-def* **unfolding** *mat-row-def*[*abs-def*] **by** *auto*

definition *transpose-row*
where *transpose-row A i* = *vec-lambda* (%*j*. *A* $ *j* $ *i*)

lemma *transpose-row-code* [*code abstract*]:
vec-nth (*transpose-row A i*) = (%*j*. *A* $ *j* $ *i*) **by** (*metis transpose-row-def vec-lambda-beta*)

lemma *transpose-code*[*code abstract*]:
vec-nth (*transpose A*) = *transpose-row A*
by (*auto simp*: *transpose-def transpose-row-def*)

lemma [*code abstract*]: *vec-nth* (*row i A*) = (($) (*A* $ *i*)) **unfolding** *row-def* **by** *fastforce*
lemma [*code abstract*]: *vec-nth* (*column j A*) = (%*i*. *A* $ *i* $ *j*) **unfolding** *column-def* **by** *fastforce*

definition *rowvector-row v i* = *vec-lambda* (%*j*. (*v*$*j*))

lemma *rowvector-row-code* [*code abstract*]:
vec-nth (*rowvector-row v i*) = (%*j*. (*v*$*j*)) **unfolding** *rowvector-row-def* **by** *auto*

lemma [*code abstract*]: *vec-nth* (*rowvector v*) = *rowvector-row v*
unfolding *rowvector-def* **unfolding** *rowvector-row-def*[*abs-def*] **by** *auto*

definition *columnvector-row v i* = *vec-lambda* (%*j*. (*v*$*i*))

lemma *columnvector-row-code* [*code abstract*]:
vec-nth (*columnvector-row v i*) = (%*j*. (*v*$*i*)) **unfolding** *columnvector-row-def* **by** *auto*

lemma [*code abstract*]: *vec-nth* (*columnvector v*) = *columnvector-row v*

unfolding *columnvector-def* **unfolding** *columnvector-row-def*[*abs-def*] **by** *auto*

end

4 Elementary Operations over matrices

theory *Elementary-Operations*
imports
Rank-Nullity-Theorem.Fundamental-Subspaces
Code-Matrix
begin

4.1 Some previous results:

lemma *mat-1-fun*: *mat 1 $ a $ b* = (*λi j. if i=j then 1 else 0*) *a b* **unfolding** *mat-def* **by** *auto*

lemma *mat1-sum-eq*:
shows (∑ *k*∈*UNIV*. *mat* (*1*::*'a*::{*semiring-1*}) $ *s* $ *k* ∗ *mat 1* $ *k* $ *t*) = *mat 1* $ *s* $ *t*
proof (*unfold mat-def*, *auto*)
let *?f*=*λk*. (*if t* = *k then 1*::*'a else* (*0*::*'a*)) ∗ (*if k* = *t then 1*::*'a else* (*0*::*'a*))
have *univ-eq*: *UNIV* = (*UNIV* − {*t*}) ∪ {*t*} **by** *fast*
have *sum ?f UNIV* = *sum ?f* ((*UNIV* − {*t*}) ∪ {*t*}) **using** *univ-eq* **by** *simp*
also have ... = *sum ?f* (*UNIV* − {*t*}) + *sum ?f* {*t*} **by** (*rule sum.union-disjoint*, *auto*)
also have ... = *0* + *sum ?f* {*t*} **by** *auto*
also have ... = *sum ?f* {*t*} **by** *simp*
also have ... = *1* **by** *simp*
finally show *sum ?f UNIV* = *1* .
next
assume *s-not-t*: *s* ≠ *t*
let *?g*=*λk*. (*if s* = *k then 1*::*'a else 0*) ∗ (*if k* = *t then 1 else 0*)
have *sum ?g UNIV* = *sum* (*λk. 0*::*'a*) (*UNIV*::*'b set*) **by** (*rule sum.cong*, *auto simp add*: *s-not-t*)
also have ... = *0* **by** *simp*
finally show *sum ?g UNIV* = *0* .
qed

lemma *invertible-mat-n*:
fixes *n*::*'a*::{*field*}
assumes *n*: *n* ≠ *0*
shows *invertible* ((*mat n*)::*'a*^*n*^*n*)
proof (*unfold invertible-def*, *rule exI*[*of - mat* (*inverse n*)], *rule conjI*)
show *mat n* ∗∗ *mat* (*inverse n*) = (*mat 1*::*'a*^*n*^*n*)
proof (*unfold matrix-matrix-mult-def mat-def*, *vector*, *auto*)
fix *ia*::*'n*
let *?f*=(*λk*. (*if ia* = *k then n else 0*) ∗ (*if k* = *ia then inverse n else 0*))

have *UNIV-rw*: (*UNIV*::*'n set*) = *insert ia* (*UNIV*−{*ia*}) **by** *auto*
have (∑ *k*∈(*UNIV*::*'n set*). (*if ia* = *k then n else 0*) ∗ (*if k* = *ia then inverse n else 0*)) =
(∑ *k*∈*insert ia* (*UNIV*−{*ia*}). (*if ia* = *k then n else 0*) ∗ (*if k* = *ia then inverse n else 0*)) **using** *UNIV-rw* **by** *simp*
also have ... = *?f ia* + *sum ?f* (*UNIV*−{*ia*})
proof (*rule sum.insert*)
show *finite* (*UNIV* − {*ia*}) **using** *finite-UNIV* **by** *fastforce*
show *ia* ∉ *UNIV* − {*ia*} **by** *fast*
qed
also have ... = *1* **using** *right-inverse*[*OF n*] **by** *simp*
finally show (∑ *k*∈(*UNIV*::*'n set*). (*if ia* = *k then n else 0*) ∗ (*if k* = *ia then inverse n else 0*)) = (*1*::*'a*) **.**
fix *i*::*'n*
assume *i-not-ia*: *i* ≠ *ia*
show (∑ *k*∈(*UNIV*::*'n set*). (*if i* = *k then n else 0*) ∗ (*if k* = *ia then inverse n else 0*)) = *0* **by** (*rule sum.neutral*, *simp add*: *i-not-ia*)
qed
next
show *mat* (*inverse n*) ** *mat n* = ((*mat 1*)::*'a*^*'n*^*'n*)
proof (*unfold matrix-matrix-mult-def mat-def*, *vector*, *auto*)
fix *ia*::*'n*
let *?f*= (λ*k*. (*if ia* = *k then inverse n else 0*) ∗ (*if k* = *ia then n else 0*))
have *UNIV-rw*: (*UNIV*::*'n set*) = *insert ia* (*UNIV*−{*ia*}) **by** *auto*
have (∑ *k*∈(*UNIV*::*'n set*). (*if ia* = *k then inverse n else 0*) ∗ (*if k* = *ia then n else 0*)) =
(∑ *k*∈*insert ia* (*UNIV*−{*ia*}). (*if ia* = *k then inverse n else 0*) ∗ (*if k* = *ia then n else 0*)) **using** *UNIV-rw* **by** *simp*
also have ... = *?f ia* + *sum ?f* (*UNIV*−{*ia*})
proof (*rule sum.insert*)
show *finite* (*UNIV* − {*ia*}) **using** *finite-UNIV* **by** *fastforce*
show *ia* ∉ *UNIV* − {*ia*} **by** *fast*
qed
also have ... = *1* **using** *left-inverse*[*OF n*] **by** *simp*
finally show (∑ *k*∈(*UNIV*::*'n set*). (*if ia* = *k then inverse n else 0*) ∗ (*if k* = *ia then n else 0*)) = (*1*::*'a*) **.**
fix *i*::*'n*
assume *i-not-ia*: *i* ≠ *ia*
show (∑ *k*∈(*UNIV*::*'n set*). (*if i* = *k then inverse n else 0*) ∗ (*if k* = *ia then n else 0*)) = *0* **by** (*rule sum.neutral*, *simp add*: *i-not-ia*)
qed
qed

corollary *invertible-mat-1*:
shows *invertible* (*mat* (*1*::*'a*::{*field*})) **by** (*metis invertible-mat-n zero-neq-one*)

4.2 Definitions of elementary row and column operations

Definitions of elementary row operations

definition *interchange-rows* :: *'a ^'n^'m => 'm => 'm ⇒ 'a ^'n^'m*
where *interchange-rows A a b = (χ i j. if i=a then A $ b $ j else if i=b then A $ a $ j else A $ i $ j)*

definition *mult-row* :: *('a::times) ^'n^'m => 'm => 'a ⇒ 'a ^'n^'m*
where *mult-row A a q = (χ i j. if i=a then q∗(A $ a $ j) else A $ i $ j)*

definition *row-add* :: *('a::{plus, times}) ^'n^'m => 'm => 'm ⇒ 'a ⇒ 'a ^'n^'m*
where *row-add A a b q = (χ i j. if i=a then (A $ a $ j) + q∗(A $ b $ j) else A $ i $ j)*

Definitions of elementary column operations

definition *interchange-columns* :: *'a ^'n^'m => 'n => 'n ⇒ 'a ^'n^'m*
where *interchange-columns A n m = (χ i j. if j=n then A $ i $ m else if j=m then A $ i $ n else A $ i $ j)*

definition *mult-column* :: *('a::times) ^'n^'m => 'n => 'a ⇒ 'a ^'n^'m*
where *mult-column A n q = (χ i j. if j=n then (A $ i $ j)∗q else A $ i $ j)*

definition *column-add* :: *('a::{plus, times}) ^'n^'m => 'n => 'n ⇒ 'a ⇒ 'a ^'n^'m*
where *column-add A n m q = (χ i j. if j=n then ((A $ i $ n) + (A $ i $ m)∗q) else A $ i $ j)*

4.3 Properties about elementary row operations

4.3.1 Properties about interchanging rows

Properties about *interchange-rows*

lemma *interchange-same-rows*: *interchange-rows A a a = A*
unfolding *interchange-rows-def* **by** *vector*

lemma *interchange-rows-i*[*simp*]: *interchange-rows A i j $ i = A $ j*
unfolding *interchange-rows-def* **by** *vector*

lemma *interchange-rows-j*[*simp*]: *interchange-rows A i j $ j = A $ i*
unfolding *interchange-rows-def* **by** *vector*

lemma *interchange-rows-preserves*:
assumes *i ≠ a* **and** *j ≠ a*
shows *interchange-rows A i j $ a = A $ a*
using *assms* **unfolding** *interchange-rows-def* **by** *vector*

lemma *interchange-rows-mat-1*:
shows *interchange-rows (mat 1) a b ∗∗ A = interchange-rows A a b*
proof (*unfold matrix-matrix-mult-def interchange-rows-def*, *vector*, *auto*)
fix *ia*
let *?f=(λk. mat (1::'a) $ a $ k ∗ A $ k $ ia)*
have *univ-rw*:*UNIV = (UNIV−{a}) ∪ {a}* **by** *auto*

have *sum ?f UNIV = sum ?f ((UNIV−{a}) ∪ {a})* **using** *univ-rw* **by** *auto*
also have *... = sum ?f (UNIV−{a}) + sum ?f {a}*
proof (*rule sum.union-disjoint*)
show *finite (UNIV − {a})* **by** (*metis finite-code*)
show *finite {a}* **by** *simp*
show *(UNIV − {a}) ∩ {a} = {}* **by** *simp*
qed
also have *... = sum ?f {a}* **unfolding** *mat-def* **by** *auto*
also have *... = ?f a* **by** *auto*
also have *... = A $ a $ ia* **unfolding** *mat-def* **by** *auto*
finally show *(∑ k∈UNIV. mat (1::'a) $ a $ k * A $ k $ ia) = A $ a $ ia* **.**
assume *i*: *a ≠ b*
let *?g= λk. mat (1::'a) $ b $ k * A $ k $ ia*
have *univ-rw'*:*UNIV = (UNIV−{b}) ∪ {b}* **by** *auto*
have *sum ?g UNIV = sum ?g ((UNIV−{b}) ∪ {b})* **using** *univ-rw'* **by** *auto*
also have *... = sum ?g (UNIV−{b}) + sum ?g {b}* **by** (*rule sum.union-disjoint*, *auto*)
also have *... = sum ?g {b}* **unfolding** *mat-def* **by** *auto*
also have *... = ?g b* **by** *simp*
finally show *(∑ k∈UNIV. mat (1::'a) $ b $ k * A $ k $ ia) = A $ b $ ia* **unfolding** *mat-def* **by** *simp*
next
fix *i j*
assume *ib*: *i ≠ b* **and** *ia*:*i ≠ a*
let *?h=λk. mat (1::'a) $ i $ k * A $ k $ j*
have *univ-rw''*:*UNIV = (UNIV−{i}) ∪ {i}* **by** *auto*
have *sum ?h UNIV = sum ?h ((UNIV−{i}) ∪ {i})* **using** *univ-rw''* **by** *auto*
also have *... = sum ?h (UNIV−{i}) + sum ?h {i}* **by** (*rule sum.union-disjoint*, *auto*)
also have *... = sum ?h {i}* **unfolding** *mat-def* **by** *auto*
also have *... = ?h i* **by** *simp*
finally show *(∑ k∈UNIV. mat (1::'a) $ i $ k * A $ k $ j) = A $ i $ j* **unfolding** *mat-def* **by** *auto*
qed

lemma *invertible-interchange-rows*: *invertible (interchange-rows (mat 1) a b)*
proof (*unfold invertible-def*, *rule exI*[*of - interchange-rows (mat 1) a b*], *simp*, *unfold matrix-matrix-mult-def*, *vector*, *clarify*,
unfold interchange-rows-def, *vector*, *unfold mat-1-fun*, *auto+*)
fix *s t*::'*b*
assume *s-not-t*: *s ≠ t*
show *(∑ k::'b∈UNIV. (if s = k then 1::'a else (0::'a)) * (if k = t then 1::'a else if k = t then 1::'a else if k = t then 1::'a else (0::'a))) = (0::'a)*
by (*rule sum.neutral*, *simp add*: *s-not-t*)
assume *b-not-t*: *b ≠ t*
show *(∑ k∈UNIV. (if s = b then if t = k then 1::'a else (0::'a) else if s = k then 1::'a else (0::'a)) *
(if k = t then 0::'a else if k = b then 1::'a else if k = t then 1::'a else (0::'a)))
=

```
    (0::'a) by (rule sum.neutral, simp)
  assume a-not-t: a ≠ t
  show (∑k∈UNIV. (if s = a then if b = k then 1::'a else (0::'a) else if s = b
then if a = k then 1::'a else (0::'a) else if s = k then 1::'a else (0::'a)) *
    (if k = a then 0::'a else if k = b then 0::'a else if k = t then 1::'a else (0::'a)))
=
    (0::'a) by (rule sum.neutral, auto simp add: s-not-t)
next
  fix s t::'b
  assume a-noteq-t: a≠t and s-noteq-t: s ≠ t
  show (∑k∈UNIV. (if s = a then if t = k then 1::'a else (0::'a) else if s = t
then if a = k then 1::'a else (0::'a) else if s = k then 1::'a else (0::'a)) *
    (if k = a then 1::'a else if k = t then 0::'a else if k = t then 1::'a else (0::'a)))
=
    (0::'a) apply (rule sum.neutral) using s-noteq-t by fastforce
next
  fix s t::'b
  show (∑k∈UNIV. (if t = k then 1::'a else (0::'a)) * (if k = t then 1::'a else if
k = t then 1::'a else if k = t then 1::'a else (0::'a))) = (1::'a)
  proof −
    let ?f=(λk. (if t = k then 1::'a else (0::'a)) * (if k = t then 1::'a else if k = t
then 1::'a else if k = t then 1::'a else (0::'a)))
    have univ-eq: UNIV = ((UNIV − {t}) ∪ {t}) by auto
    have sum ?f UNIV = sum ?f ((UNIV − {t}) ∪ {t}) using univ-eq by simp
   also have ... = sum ?f (UNIV − {t}) + sum ?f {t} by (rule sum.union-disjoint,
auto)
    also have ... = 0 + sum ?f {t} by auto
    also have ... = sum ?f {t} by simp
    also have ... = 1 by simp
    finally show ?thesis .
  qed
next
  fix s t::'b
  assume b-noteq-t: b ≠ t
  show (∑k∈UNIV. (if b = k then 1::'a else (0::'a)) * (if k = t then 0::'a else if
k = b then 1::'a else if k = t then 1::'a else (0::'a))) = (1::'a)
  proof −
    let ?f=(λk. (if b = k then 1::'a else (0::'a)) * (if k = t then 0::'a else if k = b
then 1::'a else if k = t then 1::'a else (0::'a)))
    have univ-eq: UNIV = ((UNIV − {b}) ∪ {b}) by auto
    have sum ?f UNIV = sum ?f ((UNIV − {b}) ∪ {b}) using univ-eq by simp
   also have ... = sum ?f (UNIV − {b}) + sum ?f {b} by (rule sum.union-disjoint,
auto)
    also have ... = 0 + sum ?f {b} by auto
    also have ... = sum ?f {b} by simp
    also have ... = 1 using b-noteq-t by simp
    finally show ?thesis .
  qed
  assume a-noteq-t: a≠t
```

show $(\sum k{\in}UNIV.\ (if\ t = k\ then\ 1::'a\ else\ (0::'a)) * (if\ k = a\ then\ 0::'a\ else\ if\ k = b\ then\ 0::'a\ else\ if\ k = t\ then\ 1::'a\ else\ (0::'a))) = (1::'a)$
proof −
let *?f*=$(\lambda k.\ (if\ t = k\ then\ 1::'a\ else\ (0::'a)) * (if\ k = a\ then\ 0::'a\ else\ if\ k = b\ then\ 0::'a\ else\ if\ k = t\ then\ 1::'a\ else\ (0::'a)))$
have *univ-eq*: $UNIV = ((UNIV - \{t\}) \cup \{t\})$ **by** *auto*
have *sum ?f UNIV* = *sum ?f* $((UNIV - \{t\}) \cup \{t\})$ **using** *univ-eq* **by** *simp*
also have ... = *sum ?f* $(UNIV - \{t\})$ + *sum ?f* $\{t\}$ **by** (*rule sum.union-disjoint*, *auto*)
also have ... = *0* + *sum ?f* $\{t\}$ **by** *auto*
also have ... = *sum ?f* $\{t\}$ **by** *simp*
also have ... = *1* **using** *b-noteq-t a-noteq-t* **by** *simp*
finally show *?thesis* **.**
qed
next
fix *s t*::'*b*
assume *a-noteq-t*: $a{\neq}t$
show $(\sum k{\in}UNIV.\ (if\ a = k\ then\ 1::'a\ else\ (0::'a)) * (if\ k = a\ then\ 1::'a\ else\ if\ k = t\ then\ 0::'a\ else\ if\ k = t\ then\ 1::'a\ else\ (0::'a))) = (1::'a)$
proof −
let *?f*=$\lambda k.\ (if\ a = k\ then\ 1::'a\ else\ (0::'a)) * (if\ k = a\ then\ 1::'a\ else\ if\ k = t\ then\ 0::'a\ else\ if\ k = t\ then\ 1::'a\ else\ (0::'a))$
have *univ-eq*: $UNIV = ((UNIV - \{a\}) \cup \{a\})$ **by** *auto*
have *sum ?f UNIV* = *sum ?f* $((UNIV - \{a\}) \cup \{a\})$ **using** *univ-eq* **by** *simp*
also have ... = *sum ?f* $(UNIV - \{a\})$ + *sum ?f* $\{a\}$ **by** (*rule sum.union-disjoint*, *auto*)
also have ... = *0* + *sum ?f* $\{a\}$ **by** *auto*
also have ... = *sum ?f* $\{a\}$ **by** *simp*
also have ... = *1* **using** *a-noteq-t* **by** *simp*
finally show *?thesis* **.**
qed
qed

4.3.2 Properties about multiplying a row by a constant

Properties about *mult-row*

lemma *mult-row-mat-1*: *mult-row* (*mat 1*) *a q* ** *A* = *mult-row A a q*
proof (*unfold matrix-matrix-mult-def mult-row-def*, *vector*, *auto*)
fix *ia*
let *?f*=$\lambda k.\ q * mat\ (1::'a)\ \$\ a\ \$\ k * A\ \$\ k\ \$\ ia$
have *univ-rw*:$UNIV = (UNIV{-}\{a\}) \cup \{a\}$ **by** *auto*
have *sum ?f UNIV* = *sum ?f* $((UNIV{-}\{a\}) \cup \{a\})$ **using** *univ-rw* **by** *auto*
also have ... = *sum ?f* $(UNIV{-}\{a\})$ + *sum ?f* $\{a\}$ **by** (*rule sum.union-disjoint*, *auto*)
also have ... = *sum ?f* $\{a\}$ **unfolding** *mat-def* **by** *auto*
also have ... = *?f a* **by** *auto*
also have ... = $q * A\ \$\ a\ \$\ ia$ **unfolding** *mat-def* **by** *auto*
finally show $(\sum k{\in}UNIV.\ q * mat\ (1::'a)\ \$\ a\ \$\ k * A\ \$\ k\ \$\ ia) = q * A\ \$\ a\ \$\ ia$ **.**

fix i
assume i: $i \neq a$
let $?g$=$\lambda k.\ mat\ (1::'a)\ \$\ i\ \$\ k * A\ \$\ k\ \$\ ia$
have *univ-rw''*:$UNIV = (UNIV - \{i\}) \cup \{i\}$ **by** *auto*
have $sum\ ?g\ UNIV = sum\ ?g\ ((UNIV - \{i\}) \cup \{i\})$ **using** *univ-rw''* **by** *auto*
also have $... = sum\ ?g\ (UNIV - \{i\}) + sum\ ?g\ \{i\}$ **by** (*rule sum.union-disjoint*, *auto*)
also have $... = sum\ ?g\ \{i\}$ **unfolding** *mat-def* **by** *auto*
also have $... = ?g\ i$ **by** *simp*
finally show $(\sum k{\in}UNIV.\ mat\ (1::'a)\ \$\ i\ \$\ k * A\ \$\ k\ \$\ ia) = A\ \$\ i\ \$\ ia$ **unfolding** *mat-def* **by** *simp*
qed

lemma *invertible-mult-row*:
assumes *qk*: $q * k = 1$ **and** *kq*: $k*q=1$
shows *invertible* (*mult-row* (*mat 1*) *a q*)
proof (*unfold invertible-def*, *rule exI*[*of* - *mult-row* (*mat 1*) *a k*],*rule conjI*)
show *mult-row* (*mat* $(1::'a)$) *a q* ** *mult-row* (*mat* $(1::'a)$) *a k* = *mat* $(1::'a)$
proof (*unfold matrix-matrix-mult-def*, *vector*, *clarify*, *unfold mult-row-def*, *vector*, *unfold mat-1-fun*, *auto*)
show $(\sum ka{\in}UNIV.\ q * (if\ a = ka\ then\ 1::'a\ else\ (0::'a)) * (if\ ka = a\ then\ k * (1::'a)\ else\ if\ ka = a\ then\ 1::'a\ else\ (0::'a))) = (1::'a)$
proof −
let $?f$=$\lambda ka.\ q * (if\ a = ka\ then\ 1::'a\ else\ (0::'a)) * (if\ ka = a\ then\ k * (1::'a)\ else\ if\ ka = a\ then\ 1::'a\ else\ (0::'a))$
have *univ-eq*: $UNIV = ((UNIV - \{a\}) \cup \{a\})$ **by** *auto*
have $sum\ ?f\ UNIV = sum\ ?f\ ((UNIV - \{a\}) \cup \{a\})$ **using** *univ-eq* **by** *simp*
also have $... = sum\ ?f\ (UNIV - \{a\}) + sum\ ?f\ \{a\}$ **by** (*rule sum.union-disjoint*, *auto*)
also have $... = 0 + sum\ ?f\ \{a\}$ **by** *auto*
also have $... = sum\ ?f\ \{a\}$ **by** *simp*
also have $... = 1$ **using** *qk* **by** *simp*
finally show *?thesis* **.**
qed
next
fix *s*
assume *s-noteq-a*: $s{\neq}a$
show $(\sum ka{\in}UNIV.\ (if\ s = ka\ then\ 1::'a\ else\ (0::'a)) * (if\ ka = a\ then\ k * (1::'a)\ else\ if\ ka = a\ then\ 1::'a\ else\ 0)) = 0$
by (*rule sum.neutral*, *simp add*: *s-noteq-a*)
next
fix *t*
assume *a-noteq-t*: $a{\neq}t$
show $(\sum ka{\in}UNIV.\ (if\ t = ka\ then\ 1::'a\ else\ (0::'a)) * (if\ ka = a\ then\ k * (0::'a)\ else\ if\ ka = t\ then\ 1::'a\ else\ (0::'a))) = (1::'a)$
proof −
let $?f$=$\lambda ka.\ (if\ t = ka\ then\ 1::'a\ else\ (0::'a)) * (if\ ka = a\ then\ k * (0::'a)\ else\ if\ ka = t\ then\ 1::'a\ else\ (0::'a))$
have *univ-eq*: $UNIV = ((UNIV - \{t\}) \cup \{t\})$ **by** *auto*

have *sum ?f UNIV = sum ?f ((UNIV − {t}) ∪ {t})* **using** *univ-eq* **by** *simp*
also have *... = sum ?f (UNIV − {t}) + sum ?f {t}* **by** (*rule sum.union-disjoint*, *auto*)
also have *... = sum ?f {t}* **by** *simp*
also have *... = 1* **using** *a-noteq-t* **by** *auto*
finally show *?thesis* **.**
qed
fix *s*
assume *s-not-t*: *s≠t*
show *(∑ ka∈UNIV. (if s = a then q * (if a = ka then 1::'a else (0::'a)) else if s = ka then 1::'a else (0::'a)) * (if ka = a then k * (0::'a) else if ka = t then 1::'a else (0::'a))) = (0::'a)*
by (*rule sum.neutral*, *simp add*: *s-not-t a-noteq-t*)
qed
show *mult-row (mat (1::'a)) a k ** mult-row (mat (1::'a)) a q = mat (1::'a)*
proof (*unfold matrix-matrix-mult-def*, *vector*, *clarify*, *unfold mult-row-def*, *vector*, *unfold mat-1-fun*, *auto*)
show *(∑ ka∈UNIV. k * (if a = ka then 1::'a else (0::'a)) * (if ka = a then q * (1::'a) else if ka = a then 1::'a else (0::'a))) = (1::'a)*
proof −
let *?f=λka. k * (if a = ka then 1::'a else (0::'a)) * (if ka = a then q * (1::'a) else if ka = a then 1::'a else (0::'a))*
have *univ-eq*: *UNIV = ((UNIV − {a}) ∪ {a})* **by** *auto*
have *sum ?f UNIV = sum ?f ((UNIV − {a}) ∪ {a})* **using** *univ-eq* **by** *simp*
also have *... = sum ?f (UNIV − {a}) + sum ?f {a}* **by** (*rule sum.union-disjoint*, *auto*)
also have *... = 0 + sum ?f {a}* **by** *auto*
also have *... = sum ?f {a}* **by** *simp*
also have *... = 1* **using** *kq* **by** *simp*
finally show *?thesis* **.**
qed
next
fix *s*
assume *s-not-a*: *s≠a*
show *(∑ k∈UNIV. (if s = k then 1::'a else (0::'a)) * (if k = a then q * (1::'a) else if k = a then 1::'a else (0::'a))) = (0::'a)*
by (*rule sum.neutral*, *simp add*: *s-not-a*)
next
fix *t*
assume *a-not-t*: *a≠t*
show *(∑ k∈UNIV. (if t = k then 1::'a else (0::'a)) * (if k = a then q * (0::'a) else if k = t then 1::'a else (0::'a))) = (1::'a)*
proof −
let *?f=λk. (if t = k then 1::'a else (0::'a)) * (if k = a then q * (0::'a) else if k = t then 1::'a else (0::'a))*
have *univ-eq*: *UNIV = ((UNIV − {t}) ∪ {t})* **by** *auto*
have *sum ?f UNIV = sum ?f ((UNIV − {t}) ∪ {t})* **using** *univ-eq* **by** *simp*
also have *... = sum ?f (UNIV − {t}) + sum ?f {t}* **by** (*rule sum.union-disjoint*, *auto*)

also have ... = *sum ?f* {*t*} **by** *simp*
also have ... = *1* **using** *a-not-t* **by** *simp*
finally show *?thesis* **.**
qed
fix *s*
assume *s-not-t*: *s≠t*
show (∑ *ka*∈*UNIV*. (*if s* = *a then k* ∗ (*if a* = *ka then 1*::′*a else* (*0*::′*a*)) *else if s* = *ka then 1*::′*a else* (*0*::′*a*)) ∗
(*if ka* = *a then q* ∗ (*0*::′*a*) *else if ka* = *t then 1*::′*a else* (*0*::′*a*))) = (*0*::′*a*)
by (*rule sum.neutral*, *simp add*: *s-not-t*)
qed
qed

corollary *invertible-mult-row′*:
assumes *q-not-zero*: *q* ≠ *0*
shows *invertible* (*mult-row* (*mat* (*1*::′*a*::{*field*})) *a q*)
by (*simp add*: *invertible-mult-row*[*of q inverse q*] *q-not-zero*)

4.3.3 Properties about adding a row multiplied by a constant to another row

Properties about *row-add*

lemma *row-add-mat-1*: *row-add* (*mat 1*) *a b q* ∗∗ *A* = *row-add A a b q*
proof (*unfold matrix-matrix-mult-def row-add-def*, *vector*, *auto*)
fix *j*
let *?f*= (λ*k*. (*mat* (*1*::′*a*) $ *a* $ *k* + *q* ∗ *mat* (*1*::′*a*) $ *b* $ *k*) ∗ *A* $ *k* $ *j*)
show *sum ?f UNIV* = *A* $ *a* $ *j* + *q* ∗ *A* $ *b* $ *j*
proof (*cases a*=*b*)
case *False*
have *univ-rw*: *UNIV* = {*a*} ∪ ({*b*} ∪ (*UNIV* − {*a*} − {*b*})) **by** *auto*
have *sum-rw*: *sum ?f* ({*b*} ∪ (*UNIV* − {*a*} − {*b*})) = *sum ?f* {*b*} + *sum ?f* (*UNIV* − {*a*} − {*b*}) **by** (*rule sum.union-disjoint*, *auto simp add*: *False*)
have *sum ?f UNIV* = *sum ?f* ({*a*} ∪ ({*b*} ∪ (*UNIV* − {*a*} − {*b*}))) **using** *univ-rw* **by** *simp*
also have ... = *sum ?f* {*a*} + *sum ?f* ({*b*} ∪ (*UNIV* − {*a*} − {*b*})) **by** (*rule sum.union-disjoint*, *auto simp add*: *False*)
also have ... = *sum ?f* {*a*} + *sum ?f* {*b*} + *sum ?f* (*UNIV* − {*a*} − {*b*}) **unfolding** *sum-rw add.assoc* **..**
also have ... = *sum ?f* {*a*} + *sum ?f* {*b*}
proof −
have *sum ?f* (*UNIV* − {*a*} − {*b*}) = *sum* (λ*k*. *0*) (*UNIV* − {*a*} − {*b*}) **unfolding** *mat-def* **by** (*rule sum.cong*, *auto*)
also have ... = *0* **unfolding** *sum.neutral-const* **..**
finally show *?thesis* **by** *simp*
qed
also have ... = *A* $ *a* $ *j* + *q* ∗ *A* $ *b* $ *j* **using** *False* **unfolding** *mat-def* **by** *simp*
finally show *?thesis* **.**
next

case *True*
have *univ-rw*: $UNIV = \{b\} \cup (UNIV - \{b\})$ **by** *auto*
have *sum ?f UNIV = sum ?f* $(\{b\} \cup (UNIV - \{b\}))$ **using** *univ-rw* **by** *simp*
also have ... = *sum ?f* $\{b\}$ + *sum ?f* $(UNIV - \{b\})$ **by** (*rule sum.union-disjoint*, *auto*)
also have ... = *sum ?f* $\{b\}$
proof −
have *sum ?f* $(UNIV - \{b\})$ = *sum* $(\lambda k.\ 0)$ $(UNIV - \{b\})$ **using** *True* **unfolding** *mat-def* **by** *auto*
also have ... = *0* **unfolding** *sum.neutral-const* **..**
finally show *?thesis* **by** *simp*
qed
also have ... = $A\ \$\ a\ \$\ j + q * A\ \$\ b\ \$\ j$
by (*unfold True mat-def*, *simp*, *metis* (*opaque-lifting*, *no-types*) *vector-add-component vector-sadd-rdistrib vector-smult-component vector-smult-lid*)
finally show *?thesis* **.**
qed
fix *i* **assume** *i*: $i \neq a$
let *?g*=$\lambda k.$ *mat* $(1::'a)\ \$\ i\ \$\ k * A\ \$\ k\ \$\ j$
have *univ-rw*: $UNIV = \{i\} \cup (UNIV - \{i\})$ **by** *auto*
have *sum ?g UNIV = sum ?g* $(\{i\} \cup (UNIV - \{i\}))$ **using** *univ-rw* **by** *simp*
also have ... = *sum ?g* $\{i\}$ + *sum ?g* $(UNIV - \{i\})$ **by** (*rule sum.union-disjoint*, *auto*)
also have ... = *sum ?g* $\{i\}$
proof −
have *sum ?g* $(UNIV - \{i\})$ = *sum* $(\lambda k.\ 0)$ $(UNIV - \{i\})$ **unfolding** *mat-def* **by** *auto*
also have ... = *0* **unfolding** *sum.neutral-const* **..**
finally show *?thesis* **by** *simp*
qed
also have ... = $A\ \$\ i\ \$\ j$ **unfolding** *mat-def* **by** *simp*
finally show $(\sum k \in UNIV.\ mat\ (1::'a)\ \$\ i\ \$\ k * A\ \$\ k\ \$\ j) = A\ \$\ i\ \$\ j$ **.**
qed

lemma *invertible-row-add*:
assumes *a-noteq-b*: $a \neq b$
shows *invertible* (*row-add* (*mat* $(1::'a::\{ring\text{-}1\})$) *a b q*)
proof (*unfold invertible-def*, *rule exI*[*of* - (*row-add* (*mat 1*) *a b* $(-q)$)], *rule conjI*)
show *row-add* (*mat* $(1::'a)$) *a b q* ∗∗ *row-add* (*mat* $(1::'a)$) *a b* $(-\ q)$ = *mat* $(1::'a)$ **using** *a-noteq-b*
proof (*unfold matrix-matrix-mult-def*, *vector*, *clarify*, *unfold row-add-def*, *vector*, *unfold mat-1-fun*, *auto*)
show $(\sum k::'b \in UNIV.$ (*if b = k then* $1::'a$ *else* $(0::'a)$) ∗ (*if k = a then* $(0::'a)$ + − *q* ∗ $(1::'a)$ *else if k = b then* $1::'a$ *else* $(0::'a)$)) = $(1::'a)$
proof −
let *?f*=$\lambda k.$ (*if b = k then* $1::'a$ *else* $(0::'a)$) ∗ (*if k = a then* $(0::'a)$ + − *q* ∗ $(1::'a)$ *else if k = b then* $1::'a$ *else* $(0::'a)$)
have *univ-eq*: $UNIV = ((UNIV - \{b\}) \cup \{b\})$ **by** *auto*
have *sum ?f UNIV = sum ?f* $((UNIV - \{b\}) \cup \{b\})$ **using** *univ-eq* **by** *simp*

also have ... = *sum ?f* (*UNIV* − {*b*}) + *sum ?f* {*b*} **by** (*rule sum.union-disjoint*, *auto*)
also have ... = *0* + *sum ?f* {*b*} **by** *auto*
also have ... = *sum ?f* {*b*} **by** *simp*
also have ... = *1* **using** *a-noteq-b* **by** *simp*
finally show *?thesis* **.**
qed
show (∑ *k*::′*b*∈*UNIV*. ((*if a* = *k then 1*::′*a else* (*0*::′*a*)) + *q* ∗ (*if b* = *k then 1*::′*a else* (*0*::′*a*))) ∗ (*if k* = *a then* (*1*::′*a*) + −
q ∗ (*0*::′*a*) *else if k* = *a then 1*::′*a else* (*0*::′*a*))) = (*1*::′*a*)
proof −
let *?f*=λ*k*. ((*if a* = *k then 1*::′*a else* (*0*::′*a*)) + *q* ∗ (*if b* = *k then 1*::′*a else* (*0*::′*a*))) ∗ (*if k* = *a then* (*1*::′*a*) + −
q ∗ (*0*::′*a*) *else if k* = *a then 1*::′*a else* (*0*::′*a*))
have *univ-eq*: *UNIV* = ((*UNIV* − {*a*}) ∪ {*a*}) **by** *auto*
have *sum ?f UNIV* = *sum ?f* ((*UNIV* − {*a*}) ∪ {*a*}) **using** *univ-eq* **by** *simp*
also have ... = *sum ?f* (*UNIV* − {*a*}) + *sum ?f* {*a*} **by** (*rule sum.union-disjoint*, *auto*)
also have ... = *0* + *sum ?f* {*a*} **by** *auto*
also have ... = *sum ?f* {*a*} **by** *simp*
also have ... = *1* **using** *a-noteq-b* **by** *simp*
finally show *?thesis* **.**
qed
next
fix *s*
assume *s-not-a*: *s* ≠ *a*
show (∑ *k*::′*b*∈*UNIV*. (*if s* = *k then 1*::′*a else* (*0*::′*a*)) ∗ (*if k* = *a then* (*1*::′*a*) + − *q* ∗ (*0*::′*a*) *else if k* = *a then 1*::′*a else* (*0*::′*a*))) = (*0*::′*a*)
by (*rule sum.neutral*, *auto simp add*: *s-not-a*)
next
fix *t*
assume *b-not-t*: *b* ≠ *t* **and** *a-not-t*: *a* ≠ *t*
show (∑ *k*∈*UNIV*. (*if t* = *k then 1*::′*a else* (*0*::′*a*)) ∗ (*if k* = *a then* (*0*::′*a*) + − *q* ∗ (*0*::′*a*) *else if k* = *t then 1*::′*a else* (*0*::′*a*))) = (*1*::′*a*)
proof −
let *?f*=λ*k*. (*if t* = *k then 1*::′*a else* (*0*::′*a*)) ∗ (*if k* = *a then* (*0*::′*a*) + − *q* ∗ (*0*::′*a*) *else if k* = *t then 1*::′*a else* (*0*::′*a*))
have *univ-eq*: *UNIV* = ((*UNIV* − {*t*}) ∪ {*t*}) **by** *auto*
have *sum ?f UNIV* = *sum ?f* ((*UNIV* − {*t*}) ∪ {*t*}) **using** *univ-eq* **by** *simp*
also have ... = *sum ?f* (*UNIV* − {*t*}) + *sum ?f* {*t*} **by** (*rule sum.union-disjoint*, *auto*)
also have ... = *0* + *sum ?f* {*t*} **by** *auto*
also have ... = *sum ?f* {*t*} **by** *simp*
also have ... = *1* **using** *b-not-t a-not-t* **by** *simp*
finally show *?thesis* **.**
qed
next
fix *s t*
assume *b-not-t*: *b* ≠ *t* **and** *a-not-t*: *a* ≠ *t* **and** *s-not-t*: *s* ≠ *t*

show $(\sum k{\in}UNIV.\ (if\ s = a\ then\ (if\ a = k\ then\ 1::'a\ else\ (0::'a)) + q * (if\ b = k\ then\ 1::'a\ else\ (0::'a))\ else\ if\ s = k\ then\ 1::'a\ else\ (0::'a)) *$
$(if\ k = a\ then\ (0::'a) + -\ q * (0::'a)\ else\ if\ k = t\ then\ 1::'a\ else\ (0::'a))) = (0::'a)$ **by** (*rule sum.neutral*, *auto simp add*: *b-not-t a-not-t s-not-t*)
next
fix *s*
assume *s-not-b*: $s{\neq}b$
let $?f{=}\lambda k.\ (if\ s = a\ then\ (if\ a = k\ then\ 1::'a\ else\ (0::'a)) + q * (if\ b = k\ then\ 1::'a\ else\ (0::'a))\ else\ if\ s = k\ then\ 1::'a\ else\ (0::'a)) *$
$(if\ k = a\ then\ (0::'a) + -\ q * (1::'a)\ else\ if\ k = b\ then\ 1::'a\ else\ (0::'a))$
show $sum\ ?f\ UNIV = (0::'a)$
proof (*cases* $s{=}a$)
case *False*
show *?thesis* **by** (*rule sum.neutral*, *auto simp add*: *False s-not-b a-noteq-b*)
next
case *True* — This case is different from the other cases
have *univ-eq*: $UNIV = ((UNIV - \{a\} - \{b\}) \cup (\{b\} \cup \{a\}))$ **by** *auto*
have *sum-a*: $sum\ ?f\ \{a\} = -q$ **unfolding** *True* **using** *s-not-b* **using** *a-noteq-b* **by** *auto*
have *sum-b*: $sum\ ?f\ \{b\} = q$ **unfolding** *True* **using** *s-not-b* **using** *a-noteq-b* **by** *auto*
have *sum-rest*: $sum\ ?f\ (UNIV - \{a\} - \{b\}) = 0$ **by** (*rule sum.neutral*, *auto simp add*: *True s-not-b a-noteq-b*)
have $sum\ ?f\ UNIV = sum\ ?f\ ((UNIV - \{a\} - \{b\}) \cup (\{b\} \cup \{a\}))$ **using** *univ-eq* **by** *simp*
also have $... = sum\ ?f\ (UNIV - \{a\} - \{b\}) + sum\ ?f\ (\{b\} \cup \{a\})$ **by** (*rule sum.union-disjoint*, *auto*)
also have $... = sum\ ?f\ (UNIV - \{a\} - \{b\}) + sum\ ?f\ \{b\} + sum\ ?f\ \{a\}$ **by** (*auto simp add*: *sum.union-disjoint a-noteq-b*)
also have $... = 0$ **unfolding** *sum-a sum-b sum-rest* **by** *simp*
finally show *?thesis* **.**
qed
qed
next
show $row\text{-}add\ (mat\ (1::'a))\ a\ b\ (-\ q)\ **\ row\text{-}add\ (mat\ (1::'a))\ a\ b\ q = mat\ (1::'a)$ **using** *a-noteq-b*
proof (*unfold matrix-matrix-mult-def*, *vector*, *clarify*, *unfold row-add-def*, *vector*, *unfold mat-1-fun*, *auto*)
show $(\sum k{\in}UNIV.\ (if\ b = k\ then\ 1::'a\ else\ (0::'a)) * (if\ k = a\ then\ (0::'a) + q * (1::'a)\ else\ if\ k = b\ then\ 1::'a\ else\ (0::'a))) = (1::'a)$
proof −
let $?f{=}\lambda k.\ (if\ b = k\ then\ 1::'a\ else\ (0::'a)) * (if\ k = a\ then\ (0::'a) + q * (1::'a)\ else\ if\ k = b\ then\ 1::'a\ else\ (0::'a))$
have *univ-eq*: $UNIV = ((UNIV - \{b\}) \cup \{b\})$ **by** *auto*
have $sum\ ?f\ UNIV = sum\ ?f\ ((UNIV - \{b\}) \cup \{b\})$ **using** *univ-eq* **by** *simp*
also have $... = sum\ ?f\ (UNIV - \{b\}) + sum\ ?f\ \{b\}$ **by** (*rule sum.union-disjoint*, *auto*)
also have $... = 0 + sum\ ?f\ \{b\}$ **by** *auto*
also have $... = sum\ ?f\ \{b\}$ **by** *simp*

```
    also have ... = 1 using a-noteq-b by simp
    finally show ?thesis .
  qed
 next
  show (∑ k∈UNIV. ((if a = k then 1 else 0) − q * (if b = k then 1 else 0)) *
(if k = a then 1 + q * 0 else if k = a then 1 else 0)) = 1
  proof −
   let ?f=λk. ((if a = k then 1::'a else (0::'a)) + − (q * (if b = k then 1::'a else
(0::'a)))) * (if k = a then (1::'a) + q * (0::'a) else if k = a then 1::'a else (0::'a))
    have univ-eq: UNIV = ((UNIV − {a}) ∪ {a}) by auto
   have sum ?f UNIV = sum ?f ((UNIV − {a}) ∪ {a}) using univ-eq by simp
  also have ... = sum ?f (UNIV − {a}) + sum ?f {a} by (rule sum.union-disjoint,
auto)
    also have ... = 0 + sum ?f {a} by auto
    also have ... = sum ?f {a} by simp
    also have ... = 1 using a-noteq-b by simp
    finally show ?thesis by simp
  qed
 next
  fix s
  assume s-not-a: s≠a
  show (∑ k∈UNIV. (if s = k then 1::'a else (0::'a)) * (if k = a then (1::'a) +
q * (0::'a) else if k = a then 1::'a else (0::'a))) = (0::'a)
    by (rule sum.neutral, auto simp add: s-not-a)
 next
  fix t
  assume b-not-t: b ≠ t and a-not-t: a ≠ t
  show (∑ k∈UNIV. (if t = k then 1::'a else (0::'a)) * (if k = a then (0::'a) +
q * (0::'a) else if k = t then 1::'a else (0::'a))) = (1::'a)
  proof −
    let ?f=λk. (if t = k then 1::'a else (0::'a)) * (if k = a then (0::'a) + q *
(0::'a) else if k = t then 1::'a else (0::'a))
    have univ-eq: UNIV = ((UNIV − {t}) ∪ {t}) by auto
   have sum ?f UNIV = sum ?f ((UNIV − {t}) ∪ {t}) using univ-eq by simp
  also have ... = sum ?f (UNIV − {t}) + sum ?f {t} by (rule sum.union-disjoint,
auto)
    also have ... = 0 + sum ?f {t} by auto
    also have ... = sum ?f {t} by simp
    also have ... = 1 using b-not-t a-not-t by simp
    finally show ?thesis .
  qed
 next
  fix s t
  assume b-not-t: b ≠ t and a-not-t: a ≠ t and s-not-t: s ≠ t
  show (∑ k∈UNIV. (if s = a then (if a = k then 1::'a else (0::'a)) + − q * (if
b = k then 1::'a else (0::'a)) else if s = k then 1::'a else (0::'a)) *
    (if k = a then (0::'a) + q * (0::'a) else if k = t then 1::'a else (0::'a))) =
(0::'a)
    by (rule sum.neutral, auto simp add: b-not-t a-not-t s-not-t)
```

next
fix *s*
assume *s-not-b*: *s≠b*
let *?f*=λ*k*.(*if s* = *a then* (*if a* = *k then* 1::′*a else* (0::′*a*)) + − *q* ∗ (*if b* = *k then* 1::′*a else* (0::′*a*)) *else if s* = *k then* 1::′*a else* (0::′*a*))
∗ (*if k* = *a then* (0::′*a*) + *q* ∗ (1::′*a*) *else if k* = *b then* 1::′*a else* (0::′*a*))
show *sum ?f UNIV* = *0*
proof (*cases s*=*a*)
case *False*
show *?thesis* **by** (*rule sum.neutral*, *auto simp add*: *False s-not-b a-noteq-b*)
next
case *True* — This case is different from the other cases
have *univ-eq*: *UNIV* = ((*UNIV* − {*a*}− {*b*}) ∪ ({*b*} ∪ {*a*})) **by** *auto*
have *sum-a*: *sum ?f* {*a*} = *q* **unfolding** *True* **using** *s-not-b* **using** *a-noteq-b* **by** *auto*
have *sum-b*: *sum ?f* {*b*} = −*q* **unfolding** *True* **using** *s-not-b* **using** *a-noteq-b* **by** *auto*
have *sum-rest*: *sum ?f* (*UNIV* − {*a*} − {*b*}) = *0* **by** (*rule sum.neutral*, *auto simp add*: *True s-not-b a-noteq-b*)
have *sum ?f UNIV* = *sum ?f* ((*UNIV* − {*a*}− {*b*}) ∪ ({*b*} ∪ {*a*})) **using** *univ-eq* **by** *simp*
also have ... = *sum ?f* (*UNIV* − {*a*} − {*b*}) + *sum ?f* ({*b*} ∪ {*a*}) **by** (*rule sum.union-disjoint*, *auto*)
also have ... = *sum ?f* (*UNIV* − {*a*} − {*b*}) + *sum ?f* {*b*} + *sum ?f* {*a*} **by** (*auto simp add*: *sum.union-disjoint a-noteq-b*)
also have ... = *0* **unfolding** *sum-a sum-b sum-rest* **by** *simp*
finally show *?thesis* **.**
qed
qed
qed

4.4 Properties about elementary column operations

4.4.1 Properties about interchanging columns

Properties about *interchange-columns*

lemma *interchange-columns-mat-1*: *A* ∗∗ *interchange-columns* (*mat 1*) *a b* = *interchange-columns A a b*
proof (*unfold matrix-matrix-mult-def*, *unfold interchange-columns-def*, *vector*, *auto*)

fix *i*
show (∑ *k*∈*UNIV*. *A* $ *i* $ *k* ∗ *mat* (1::′*a*) $ *k* $ *a*) = *A* $ *i* $ *a*
proof −
let *?f*=(λ*k*. *A* $ *i* $ *k* ∗ *mat* (1::′*a*) $ *k* $ *a*)
have *univ-rw*:*UNIV* = (*UNIV*−{*a*}) ∪ {*a*} **by** *auto*
have *sum ?f UNIV* = *sum ?f* ((*UNIV*−{*a*}) ∪ {*a*}) **using** *univ-rw* **by** *auto*
also have ... = *sum ?f* (*UNIV*−{*a*}) + *sum ?f* {*a*} **by** (*rule sum.union-disjoint*, *auto*)
also have ... = *sum ?f* {*a*} **unfolding** *mat-def* **by** *auto*

finally show *?thesis* **unfolding** *mat-def* **by** *simp*
qed
assume *a-not-b*: $a \neq b$
show $(\sum k \in UNIV.\ A\ \$\ i\ \$\ k * mat\ (1::'a)\ \$\ k\ \$\ b) = A\ \$\ i\ \$\ b$
proof −
let *?f*=$(\lambda k.\ A\ \$\ i\ \$\ k * mat\ (1::'a)\ \$\ k\ \$\ b)$
have *univ-rw*:$UNIV = (UNIV - \{b\}) \cup \{b\}$ **by** *auto*
have *sum ?f UNIV* = *sum ?f* $((UNIV - \{b\}) \cup \{b\})$ **using** *univ-rw* **by** *auto*
also have ... = *sum ?f* $(UNIV - \{b\})$ + *sum ?f* $\{b\}$ **by** (*rule sum.union-disjoint*, *auto*)
also have ... = *sum ?f* $\{b\}$ **unfolding** *mat-def* **by** *auto*
finally show *?thesis* **unfolding** *mat-def* **by** *simp*
qed
next
fix *i j*
assume *j-not-b*: $j \neq b$ **and** *j-not-a*: $j \neq a$
show $(\sum k \in UNIV.\ A\ \$\ i\ \$\ k * mat\ (1::'a)\ \$\ k\ \$\ j) = A\ \$\ i\ \$\ j$
proof −
let *?f*=$(\lambda k.\ A\ \$\ i\ \$\ k * mat\ (1::'a)\ \$\ k\ \$\ j)$
have *univ-rw*:$UNIV = (UNIV - \{j\}) \cup \{j\}$ **by** *auto*
have *sum ?f UNIV* = *sum ?f* $((UNIV - \{j\}) \cup \{j\})$ **using** *univ-rw* **by** *auto*
also have ... = *sum ?f* $(UNIV - \{j\})$ + *sum ?f* $\{j\}$ **by** (*rule sum.union-disjoint*, *auto*)
also have ... = *sum ?f* $\{j\}$ **unfolding** *mat-def* **using** *j-not-b j-not-a* **by** *auto*
finally show *?thesis* **unfolding** *mat-def* **by** *simp*
qed
qed

lemma *invertible-interchange-columns*: *invertible* (*interchange-columns* (*mat 1*) *a b*)
proof (*unfold invertible-def*, *rule exI*[*of - interchange-columns* (*mat 1*) *a b*], *simp*, *unfold matrix-matrix-mult-def*, *vector*, *clarify*,
unfold interchange-columns-def, *vector*, *unfold mat-1-fun*, *auto*+)
show $(\sum k \in UNIV.\ (if\ k = b\ then\ 1::'a\ else\ if\ k = b\ then\ 1::'a\ else\ if\ b = k\ then\ 1::'a\ else\ (0::'a)) * (if\ k = b\ then\ 1::'a\ else\ (0::'a))) = (1::'a)$
proof −
let *?f*=$(\lambda k.\ (if\ k = b\ then\ 1::'a\ else\ if\ k = b\ then\ 1::'a\ else\ if\ b = k\ then\ 1::'a\ else\ (0::'a)) * (if\ k = b\ then\ 1::'a\ else\ (0::'a)))$
have *univ-rw*:$UNIV = (UNIV - \{b\}) \cup \{b\}$ **by** *auto*
have *sum ?f UNIV* = *sum ?f* $((UNIV - \{b\}) \cup \{b\})$ **using** *univ-rw* **by** *auto*
also have ... = *sum ?f* $(UNIV - \{b\})$ + *sum ?f* $\{b\}$ **by** (*rule sum.union-disjoint*, *auto*)
also have ... = *sum ?f* $\{b\}$ **by** *auto*
finally show *?thesis* **by** *simp*
qed
assume *a-not-b*: $a \neq b$
show $(\sum k \in UNIV.\ (if\ k = a\ then\ 0::'a\ else\ if\ k = b\ then\ 1::'a\ else\ if\ a = k\ then\ 1::'a\ else\ (0::'a)) * (if\ k = b\ then\ 1::'a\ else\ (0::'a))) = (1::'a)$
proof −

let *?f=λk. (if k = a then 0::'a else if k = b then 1::'a else if a = k then 1::'a else (0::'a)) ∗ (if k = b then 1::'a else (0::'a))*
have *univ-rw:UNIV = (UNIV−{b}) ∪ {b}* **by** *auto*
have *sum ?f UNIV = sum ?f ((UNIV−{b}) ∪ {b})* **using** *univ-rw* **by** *auto*
also have *... = sum ?f (UNIV−{b}) + sum ?f {b}* **by** *(rule sum.union-disjoint, auto)*
also have *... = sum ?f {b}* **using** *a-not-b* **by** *simp*
finally show *?thesis* **using** *a-not-b* **by** *auto*
qed
next
fix *t*
assume *b-not-t: b ≠ t*
show *(∑ k∈UNIV. (if k = b then 1::'a else if k = b then 1::'a else if b = k then 1::'a else (0::'a)) ∗ (if k = t then 1::'a else (0::'a))) = (0::'a)*
apply *(rule sum.neutral)* **using** *b-not-t* **by** *auto*
assume *b-not-a: b ≠ a*
show *(∑ k∈UNIV. (if k = a then 1::'a else if k = b then 0::'a else if b = k then 1::'a else (0::'a)) ∗*
(if t = a then if k = b then 1::'a else (0::'a) else if t = b then if k = a then 1::'a else (0::'a) else if k = t then 1::'a else (0::'a))) =
(0::'a) **apply** *(rule sum.neutral)* **using** *b-not-t* **by** *auto*
next
fix *t*
assume *a-not-b: a ≠ b* **and** *a-not-t: a ≠ t*
show *(∑ k∈UNIV. (if k = a then 0::'a else if k = b then 1::'a else if a = k then 1::'a else (0::'a)) ∗*
(if t = b then if k = a then 1::'a else (0::'a) else if k = t then 1::'a else (0::'a))) = (0::'a)
by *(rule sum.neutral, auto simp add: a-not-b a-not-t)*
next
assume *b-not-a: b ≠ a*
show *(∑ k∈UNIV. (if k = a then 1::'a else if k = b then 0::'a else if b = k then 1::'a else (0::'a)) ∗ (if k = a then 1::'a else (0::'a))) = (1::'a)*
proof −
let *?f=λk. (if k = a then 1::'a else if k = b then 0::'a else if b = k then 1::'a else (0::'a)) ∗ (if k = a then 1::'a else (0::'a))*
have *univ-rw:UNIV = (UNIV−{a}) ∪ {a}* **by** *auto*
have *sum ?f UNIV = sum ?f ((UNIV−{a}) ∪ {a})* **using** *univ-rw* **by** *auto*
also have *... = sum ?f (UNIV−{a}) + sum ?f {a}* **by** *(rule sum.union-disjoint, auto)*
also have *... = sum ?f {a}* **using** *b-not-a* **by** *simp*
finally show *?thesis* **using** *b-not-a* **by** *auto*
qed
next
fix *t*
assume *t-not-a: t ≠ a* **and** *t-not-b: t ≠ b*
show *(∑ k∈UNIV. (if k = a then 0::'a else if k = b then 0::'a else if t = k then 1::'a else (0::'a)) ∗ (if k = t then 1::'a else (0::'a))) = (1::'a)*
proof −

let *?f*=λk. (*if* $k = a$ *then* $0::'a$ *else if* $k = b$ *then* $0::'a$ *else if* $t = k$ *then* $1::'a$ *else* $(0::'a)$) $*$ (*if* $k = t$ *then* $1::'a$ *else* $(0::'a)$)
have *univ-rw*:*UNIV* = (*UNIV*−{t}) ∪ {t} **by** *auto*
have *sum ?f UNIV* = *sum ?f* ((*UNIV*−{t}) ∪ {t}) **using** *univ-rw* **by** *auto*
also have ... = *sum ?f* (*UNIV*−{t}) + *sum ?f* {t} **by** (*rule sum.union-disjoint*, *auto*)
also have ... = *sum ?f* {t} **using** *t-not-a t-not-b* **by** *simp*
also have ... = *1* **using** *t-not-a t-not-b* **by** *simp*
finally show *?thesis* **.**
qed
next
fix *s t*
assume *s-not-a*: $s \neq a$ **and** *s-not-b*: $s \neq b$ **and** *s-not-t*: $s \neq t$
show ($\sum k{\in}UNIV$. (*if* $k = a$ *then* $0::'a$ *else if* $k = b$ *then* $0::'a$ *else if* $s = k$ *then* $1::'a$ *else* $(0::'a)$) $*$
(*if* $t = a$ *then if* $k = b$ *then* $1::'a$ *else* $(0::'a)$ *else if* $t = b$ *then if* $k = a$ *then* $1::'a$ *else* $(0::'a)$ *else if* $k = t$ *then* $1::'a$ *else* $(0::'a)$))) =
$(0::'a)$
by (*rule sum.neutral*, *auto simp add*: *s-not-a s-not-b s-not-t*)
qed

4.4.2 Properties about multiplying a column by a constant

Properties about *mult-column*

lemma *mult-column-mat-1*: A ** *mult-column* (*mat 1*) a q = *mult-column* A a q
proof (*unfold matrix-matrix-mult-def*, *unfold mult-column-def*, *vector*, *auto*)
fix *i*
show ($\sum k{\in}UNIV$. A $ i $ k $ $(mat\ (1::'a)$ $ k $ $a * q)) = A $ i $ $a * q$
proof −
let *?f*=λk. A $ i $ k $ $(mat\ (1::'a)$ $ k $ $a * q)$
have *univ-rw*:*UNIV* = (*UNIV*−{a}) ∪ {a} **by** *auto*
have *sum ?f UNIV* = *sum ?f* ((*UNIV*−{a}) ∪ {a}) **using** *univ-rw* **by** *auto*
also have ... = *sum ?f* (*UNIV*−{a}) + *sum ?f* {a} **by** (*rule sum.union-disjoint*, *auto*)
also have ... = *sum ?f* {a} **unfolding** *mat-def* **by** *auto*
also have ... = A $ i $ $a * q$ **unfolding** *mat-def* **by** *auto*
finally show *?thesis* **.**
qed
fix *j*
show ($\sum k{\in}UNIV$. A $ i $ $k * mat\ (1::'a)$ $ k $ j) = A $ i $ j
proof −
let *?f*=λk. A $ i $ $k * mat\ (1::'a)$ $ k $ j
have *univ-rw*:*UNIV* = (*UNIV*−{j}) ∪ {j} **by** *auto*
have *sum ?f UNIV* = *sum ?f* ((*UNIV*−{j}) ∪ {j}) **using** *univ-rw* **by** *auto*
also have ... = *sum ?f* (*UNIV*−{j}) + *sum ?f* {j} **by** (*rule sum.union-disjoint*, *auto*)
also have ... = *sum ?f* {j} **unfolding** *mat-def* **by** *auto*
also have ... = A $ i $ j **unfolding** *mat-def* **by** *auto*
finally show *?thesis* **.**

qed
qed

lemma *invertible-mult-column*:
assumes *qk*: $q * k = 1$ **and** *kq*: $k * q = 1$
shows *invertible* (*mult-column* (*mat 1*) *a q*)
proof (*unfold invertible-def*, *rule exI*[*of* - *mult-column* (*mat 1*) *a k*], *rule conjI*)
show *mult-column* (*mat 1*) *a q* ** *mult-column* (*mat 1*) *a k* = *mat 1*
proof (*unfold matrix-matrix-mult-def*, *vector*, *clarify*, *unfold mult-column-def*, *vector*, *unfold mat-1-fun*, *auto*)
fix *t*
show ($\sum$ *ka*∈*UNIV*. (*if ka* = *a then* (*if t* = *ka then 1*::′*a else* (*0*::′*a*)) * *q else if t* = *ka then 1*::′*a else* (*0*::′*a*)) *
(*if t* = *a then* (*if ka* = *t then 1*::′*a else* (*0*::′*a*)) * *k else if ka* = *t then 1*::′*a else* (*0*::′*a*))) =
(*1*::′*a*)
proof −
let *?f*= λ*ka*. (*if ka* = *a then* (*if t* = *ka then 1*::′*a else* (*0*::′*a*)) * *q else if t* = *ka then 1*::′*a else* (*0*::′*a*)) *
(*if t* = *a then* (*if ka* = *t then 1*::′*a else* (*0*::′*a*)) * *k else if ka* = *t then 1*::′*a else* (*0*::′*a*))
have *univ-rw*:*UNIV* = (*UNIV*−{*t*}) ∪ {*t*} **by** *auto*
have *sum ?f UNIV* = *sum ?f* ((*UNIV*−{*t*}) ∪ {*t*}) **using** *univ-rw* **by** *auto*
also have ... = *sum ?f* (*UNIV*−{*t*}) + *sum ?f* {*t*} **by** (*rule sum.union-disjoint*, *auto*)
also have ... = *sum ?f* {*t*} **by** *auto*
also have ... = *1* **using** *qk* **by** *auto*
finally show *?thesis* **.**
qed
fix *s*
assume *s-not-t*: $s \neq t$
show ($\sum$ *ka*∈*UNIV*. (*if ka* = *a then* (*if s* = *ka then 1*::′*a else* (*0*::′*a*)) * *q else if s* = *ka then 1*::′*a else* (*0*::′*a*)) *
(*if t* = *a then* (*if ka* = *t then 1*::′*a else* (*0*::′*a*)) * *k else if ka* = *t then 1*::′*a else* (*0*::′*a*))) =
(*0*::′*a*)
apply (*rule sum.neutral*) **using** *s-not-t* **by** *auto*
qed
show *mult-column* (*mat* (*1*::′*a*)) *a k* ** *mult-column* (*mat* (*1*::′*a*)) *a q* = *mat* (*1*::′*a*)
proof (*unfold matrix-matrix-mult-def*, *vector*, *clarify*, *unfold mult-column-def*, *vector*, *unfold mat-1-fun*, *auto*)
fix *t*
show ($\sum$ *ka*∈*UNIV*. (*if ka* = *a then* (*if t* = *ka then 1*::′*a else* (*0*::′*a*)) * *k else if t* = *ka then 1*::′*a else* (*0*::′*a*)) *
(*if t* = *a then* (*if ka* = *t then 1*::′*a else* (*0*::′*a*)) * *q else if ka* = *t then 1*::′*a else* (*0*::′*a*))) = (*1*::′*a*)
proof −
let *?f*= λ*ka*. (*if ka* = *a then* (*if t* = *ka then 1*::′*a else* (*0*::′*a*)) * *k else if t* =

ka then 1::'a else (0::'a)) ∗
(if t = a then (if ka = t then 1::'a else (0::'a)) ∗ q else if ka = t then 1::'a else (0::'a))
have *univ-rw:UNIV = (UNIV−{t}) ∪ {t}* **by** *auto*
have *sum ?f UNIV = sum ?f ((UNIV−{t}) ∪ {t})* **using** *univ-rw* **by** *auto*
also have *... = sum ?f (UNIV−{t}) + sum ?f {t}* **by** (*rule sum.union-disjoint*, *auto*)
also have *... = sum ?f {t}* **by** *auto*
also have *... = 1* **using** *kq* **by** *auto*
finally show *?thesis* **.**
qed
fix *s* **assume** *s-not-t*: *s ≠ t*
show *(∑ ka∈UNIV. (if ka = a then (if s = ka then 1::'a else (0::'a)) ∗ k else if s = ka then 1::'a else (0::'a)) ∗*
(if t = a then (if ka = t then 1::'a else (0::'a)) ∗ q else if ka = t then 1::'a else (0::'a))) = 0
apply (*rule sum.neutral*) **using** *s-not-t* **by** *auto*
qed
qed

corollary *invertible-mult-column'*:
assumes *q-not-zero*: *q ≠ 0*
shows *invertible* (*mult-column* (*mat* (*1*::'*a*::{*field*})) *a q*)
by (*simp add*: *invertible-mult-column*[*of q inverse q*] *q-not-zero*)

4.4.3 Properties about adding a column multiplied by a constant to another column

Properties about *column-add*

lemma *column-add-mat-1*: *A ∗∗ column-add* (*mat 1*) *a b q = column-add A a b q*
proof (*unfold matrix-matrix-mult-def*,
unfold column-add-def, *vector*, *auto*)
fix *i*
let *?f*=λ*k*. *A $ i $ k ∗ (mat* (*1*::'*a*) *$ k $ a + mat* (*1*::'*a*) *$ k $ b ∗ q*)
show *sum ?f UNIV = A $ i $ a + A $ i $ b ∗ q*
proof (*cases a*=*b*)
case *True*
have *univ-rw:UNIV = (UNIV−{a}) ∪ {a}* **by** *auto*
have *sum ?f UNIV = sum ?f ((UNIV−{a}) ∪ {a})* **using** *univ-rw* **by** *auto*
also have *... = sum ?f (UNIV−{a}) + sum ?f {a}* **by** (*rule sum.union-disjoint*, *auto*)
also have *... = sum ?f {a}* **unfolding** *mat-def True* **by** *auto*
also have *... = ?f a* **by** *auto*
also have *... = A $ i $ a + A $ i $ b ∗ q* **using** *True* **unfolding** *mat-1-fun* **using** *distrib-left*[*of A $ i $ b 1 q*] **by** *auto*
finally show *?thesis* **.**
next
case *False*
have *univ-rw*: *UNIV = {a} ∪ ({b} ∪ (UNIV − {a} − {b}))* **by** *auto*

have *sum-rw*: *sum ?f* ({*b*} ∪ (*UNIV* − {*a*} − {*b*})) = *sum ?f* {*b*} + *sum ?f* (*UNIV* − {*a*} − {*b*}) **by** (*rule sum.union-disjoint*, *auto simp add*: *False*)
have *sum ?f UNIV* = *sum ?f* ({*a*} ∪ ({*b*} ∪ (*UNIV* − {*a*} − {*b*}))) **using** *univ-rw* **by** *simp*
also have ... = *sum ?f* {*a*} + *sum ?f* ({*b*} ∪ (*UNIV* − {*a*} − {*b*})) **by** (*rule sum.union-disjoint*, *auto simp add*: *False*)
also have ... = *sum ?f* {*a*} + *sum ?f* {*b*} + *sum ?f* (*UNIV* − {*a*} − {*b*})
unfolding *sum-rw add.assoc*[*symmetric*] **..**
also have ... = *sum ?f* {*a*} + *sum ?f* {*b*} **unfolding** *mat-def* **by** *auto*
also have ... = *A* $ *i* $ *a* + *A* $ *i* $ *b* ∗ *q* **using** *False* **unfolding** *mat-def* **by** *simp*
finally show *?thesis* **.**
qed
fix *j*
assume *j-noteq-a*: *j*≠*a*
show (∑ *k*∈*UNIV*. *A* $ *i* $ *k* ∗ *mat* (1::′*a*) $ *k* $ *j*) = *A* $ *i* $ *j*
proof −
let *?f*=λ*k*. *A* $ *i* $ *k* ∗ *mat* (1::′*a*) $ *k* $ *j*
have *univ-rw*:*UNIV* = (*UNIV*−{*j*}) ∪ {*j*} **by** *auto*
have *sum ?f UNIV* = *sum ?f* ((*UNIV*−{*j*}) ∪ {*j*}) **using** *univ-rw* **by** *auto*
also have ... = *sum ?f* (*UNIV*−{*j*}) + *sum ?f* {*j*} **by** (*rule sum.union-disjoint*, *auto*)
also have ... = *sum ?f* {*j*} **unfolding** *mat-def* **by** *auto*
also have ... = *A* $ *i* $ *j* **unfolding** *mat-def* **by** *simp*
finally show *?thesis* **.**
qed
qed

lemma *invertible-column-add*:
assumes *a-noteq-b*: *a*≠*b*
shows *invertible* (*column-add* (*mat* (1::′*a*::{*ring-1*})) *a b q*)
proof (*unfold invertible-def*, *rule exI*[*of* - (*column-add* (*mat 1*) *a b* (−*q*))], *rule conjI*)
show *column-add* (*mat* (1::′*a*)) *a b q* ∗∗ *column-add* (*mat* (1::′*a*)) *a b* (− *q*) = *mat* (1::′*a*) **using** *a-noteq-b*
proof (*unfold matrix-matrix-mult-def*, *vector*, *clarify*, *unfold column-add-def*, *vector*, *unfold mat-1-fun*, *auto*)
show (∑ *k*∈*UNIV*. (*if k* = *a then* (0::′*a*) + (1::′*a*) ∗ *q else if b* = *k then* 1::′*a else* (0::′*a*)) ∗ (*if k* = *b then* 1::′*a else* (0::′*a*))) = (1::′*a*)
proof −
let *?f*=λ*k*. (*if k* = *a then* (0::′*a*) + (1::′*a*) ∗ *q else if b* = *k then* 1::′*a else* (0::′*a*)) ∗ (*if k* = *b then* 1::′*a else* (0::′*a*))
have *univ-rw*:*UNIV* = (*UNIV*−{*b*}) ∪ {*b*} **by** *auto*
have *sum ?f UNIV* = *sum ?f* ((*UNIV*−{*b*}) ∪ {*b*}) **using** *univ-rw* **by** *auto*
also have ... = *sum ?f* (*UNIV*−{*b*}) + *sum ?f* {*b*} **by** (*rule sum.union-disjoint*, *auto*)
also have ... = *sum ?f* {*b*} **by** *auto*
also have ... = *1* **using** *a-noteq-b* **by** *simp*

finally show *?thesis* **.**
qed
show $(\sum k{\in}UNIV.\ (if\ k = a\ then\ 1 + 0 * q\ else\ if\ a = k\ then\ 1\ else\ 0) * ((if\ k = a\ then\ 1\ else\ 0) - (if\ k = b\ then\ 1\ else\ 0) * q)) = 1$
proof −
let *?f*=$\lambda k.\ (if\ k = a\ then\ (1::'a) + (0::'a) * q\ else\ if\ a = k\ then\ 1::'a\ else\ (0::'a)) * ((if\ k = a\ then\ 1::'a\ else\ (0::'a)) + - ((if\ k = b\ then\ 1::'a\ else\ (0::'a)) * q))$
have *univ-rw*:$UNIV = (UNIV-\{a\}) \cup \{a\}$ **by** *auto*
have *sum ?f UNIV* = *sum ?f* $((UNIV-\{a\}) \cup \{a\})$ **using** *univ-rw* **by** *auto*
also have ... = *sum ?f* $(UNIV-\{a\})$ + *sum ?f* $\{a\}$ **by** (*rule sum.union-disjoint*, *auto*)
also have ... = *sum ?f* $\{a\}$ **by** *auto*
also have ... = *1* **using** *a-noteq-b* **by** *simp*
finally show *?thesis* **by** *simp*
qed
fix *i j*
assume *i-not-b*: $i \neq b$ **and** *i-not-a*: $i \neq a$ **and** *i-not-j*: $i \neq j$
show $(\sum k{\in}UNIV.\ (if\ k = a\ then\ (0::'a) + (0::'a) * q\ else\ if\ i = k\ then\ 1::'a\ else\ (0::'a)) *$
$(if\ j = a\ then\ (if\ k = a\ then\ 1::'a\ else\ (0::'a)) + (if\ k = b\ then\ 1::'a\ else\ (0::'a)) * - q\ else\ if\ k = j\ then\ 1::'a\ else\ (0::'a))) = (0::'a)$
by (*rule sum.neutral*, *auto simp add*: *i-not-b i-not-a i-not-j*)
next
fix *j*
assume *a-not-j*: $a{\neq}j$
show $(\sum k{\in}UNIV.\ (if\ k = a\ then\ (1::'a) + (0::'a) * q\ else\ if\ a = k\ then\ 1::'a\ else\ (0::'a)) * (if\ k = j\ then\ 1::'a\ else\ (0::'a))) = (0::'a)$
apply (*rule sum.neutral*) **using** *a-not-j a-noteq-b* **by** *auto*
next
fix *j*
assume *j-not-b*: $j \neq b$ **and** *j-not-a*: $j \neq a$
show $(\sum k{\in}UNIV.\ (if\ k = a\ then\ (0::'a) + (0::'a) * q\ else\ if\ j = k\ then\ 1::'a\ else\ (0::'a)) * (if\ k = j\ then\ 1::'a\ else\ (0::'a))) = (1::'a)$
proof −
let *?f*=$\lambda k.\ (if\ k = a\ then\ (0::'a) + (0::'a) * q\ else\ if\ j = k\ then\ 1::'a\ else\ (0::'a)) * (if\ k = j\ then\ 1::'a\ else\ (0::'a))$
have *univ-rw*:$UNIV = (UNIV-\{j\}) \cup \{j\}$ **by** *auto*
have *sum ?f UNIV* = *sum ?f* $((UNIV-\{j\}) \cup \{j\})$ **using** *univ-rw* **by** *auto*
also have ... = *sum ?f* $(UNIV-\{j\})$ + *sum ?f* $\{j\}$ **by** (*rule sum.union-disjoint*, *auto*)
also have ... = *sum ?f* $\{j\}$ **using** *j-not-b j-not-a* **by** *auto*
also have ... = *1* **using** *j-not-b j-not-a* **by** *auto*
finally show *?thesis* **.**
qed
next
fix *j*
assume *b-not-j*: $b \neq j$
show $(\sum k{\in}UNIV.\ (if\ k = a\ then\ 0 + 1 * q\ else\ if\ b = k\ then\ 1\ else\ 0) *$

(*if j = a then (if k = a then 1 else 0) + (if k = b then 1 else 0) * − q else if k = j then 1 else 0*)) = *0*
proof (*cases j=a*)
case *False*
show *?thesis* **by** (*rule sum.neutral*, *auto simp add*: *False b-not-j*)
next
case *True* — This case is different from the other cases
let *?f=λk.* (*if k = a then 0 + 1 * q else if b = k then 1 else 0*) *
(*if j = a then (if k = a then 1 else 0) + (if k = b then 1 else 0) * − q else if k = j then 1 else 0*)
have *univ-eq*: *UNIV* = ((*UNIV* − {*a*}− {*b*}) ∪ ({*b*} ∪ {*a*})) **by** *auto*
have *sum-a*: *sum ?f* {*a*} = *q* **unfolding** *True* **using** *b-not-j* **using** *a-noteq-b* **by** *auto*
have *sum-b*: *sum ?f* {*b*} = −*q* **unfolding** *True* **using** *b-not-j* **using** *a-noteq-b* **by** *auto*
have *sum-rest*: *sum ?f* (*UNIV* − {*a*} − {*b*}) = *0* **by** (*rule sum.neutral*, *auto simp add*: *True b-not-j a-noteq-b*)
have *sum ?f UNIV* = *sum ?f* ((*UNIV* − {*a*}− {*b*}) ∪ ({*b*} ∪ {*a*})) **using** *univ-eq* **by** *simp*
also have ... = *sum ?f* (*UNIV* − {*a*} − {*b*}) + *sum ?f* ({*b*} ∪ {*a*}) **by** (*rule sum.union-disjoint*, *auto*)
also have ... = *sum ?f* (*UNIV* − {*a*} − {*b*}) + *sum ?f* {*b*} + *sum ?f* {*a*} **by** (*auto simp add*: *sum.union-disjoint a-noteq-b*)
also have ... = *0* **unfolding** *sum-a sum-b sum-rest* **by** *simp*
finally show *?thesis* **.**
qed
qed
next
show *column-add* (*mat* (*1*::′*a*)) *a b* (− *q*) ∗∗ *column-add* (*mat* (*1*::′*a*)) *a b q* = *mat* (*1*::′*a*) **using** *a-noteq-b*
proof (*unfold matrix-matrix-mult-def*, *vector*, *clarify*, *unfold column-add-def*, *vector*, *unfold mat-1-fun*, *auto*)
show (∑ *k*∈*UNIV*. (*if k = a then* (*0*::′*a*) + (*1*::′*a*) * − *q else if b = k then 1*::′*a else* (*0*::′*a*)) * (*if k = b then 1*::′*a else* (*0*::′*a*))) = (*1*::′*a*)
proof −
let *?f=λk.* (*if k = a then* (*0*::′*a*) + (*1*::′*a*) * − *q else if b = k then 1*::′*a else* (*0*::′*a*)) * (*if k = b then 1*::′*a else* (*0*::′*a*))
have *univ-rw*:*UNIV* = (*UNIV*−{*b*}) ∪ {*b*} **by** *auto*
have *sum ?f UNIV* = *sum ?f* ((*UNIV*−{*b*}) ∪ {*b*}) **using** *univ-rw* **by** *auto*
also have ... = *sum ?f* (*UNIV*−{*b*}) + *sum ?f* {*b*} **by** (*rule sum.union-disjoint*, *auto*)
also have ... = *sum ?f* {*b*} **by** *auto*
also have ... = *1* **using** *a-noteq-b* **by** *auto*
finally show *?thesis* **.**
qed
next
show (∑ *k*∈*UNIV*. (*if k = a then* (*1*::′*a*) + (*0*::′*a*) * − *q else if a = k then 1*::′*a else* (*0*::′*a*)) * ((*if k = a then 1*::′*a else* (*0*::′*a*)) + (*if k = b then 1*::′*a else* (*0*::′*a*)) * *q*)) =

```
    (1::'a)
   proof −
     let ?f=λk. (if k = a then (1::'a) + (0::'a) ∗ − q else if a = k then 1::'a else
(0::'a)) ∗ ((if k = a then 1::'a else (0::'a)) + (if k = b then 1::'a else (0::'a)) ∗ q)
     have univ-rw:UNIV = (UNIV−{a}) ∪ {a} by auto
     have sum ?f UNIV = sum ?f ((UNIV−{a}) ∪ {a}) using univ-rw by auto
    also have ... = sum ?f (UNIV−{a}) + sum ?f {a} by (rule sum.union-disjoint,
auto)
     also have ... = sum ?f {a} by auto
     also have ... =  1 using a-noteq-b by auto
     finally show ?thesis .
   qed
  next
   fix j
   assume a-not-j: a ≠ j show (∑ k∈UNIV. (if k = a then (1::'a) + (0::'a) ∗ −
q else if a = k then 1::'a else (0::'a)) ∗ (if k = j then 1::'a else (0::'a))) = (0::'a)
     apply (rule sum.neutral) using a-not-j by auto
  next
   fix j
   assume j-not-b: j ≠ b and j-not-a: j ≠ a
    show (∑ k∈UNIV. (if k = a then (0::'a) + (0::'a) ∗ − q else if j = k then
1::'a else (0::'a)) ∗ (if k = j then 1::'a else (0::'a))) = (1::'a)
   proof −
     let ?f=λk.(if k = a then (0::'a) + (0::'a) ∗ − q else if j = k then 1::'a else
(0::'a)) ∗ (if k = j then 1::'a else (0::'a))
     have univ-rw:UNIV = (UNIV−{j}) ∪ {j} by auto
     have sum ?f UNIV = sum ?f ((UNIV−{j}) ∪ {j}) using univ-rw by auto
    also have ... = sum ?f (UNIV−{j}) + sum ?f {j} by (rule sum.union-disjoint,
auto)
     also have ... = sum ?f {j} by auto
     also have ... =  1 using a-noteq-b j-not-b j-not-a by auto
     finally show ?thesis .
   qed
  next
   fix i j
   assume i-not-b: i ≠ b and i-not-a: i ≠ a and i-not-j: i ≠ j
    show (∑ k∈UNIV. (if k = a then (0::'a) + (0::'a) ∗ − q else if i = k then
1::'a else (0::'a)) ∗
      (if j = a then (if k = a then 1::'a else (0::'a)) + (if k = b then 1::'a else
(0::'a)) ∗ q else if k = j then 1::'a else (0::'a))) = (0::'a)
     by (rule sum.neutral, auto simp add: i-not-b i-not-a i-not-j)
  next
   fix j
   assume b-not-j: b ≠ j
    show (∑ k∈UNIV. (if k = a then (0::'a) + (1::'a) ∗ − q else if b = k then
1::'a else (0::'a)) ∗
      (if j = a then (if k = a then 1::'a else (0::'a)) + (if k = b then 1::'a else
(0::'a)) ∗ q else if k = j then 1::'a else (0::'a))) = 0
   proof (cases j=a)
```

case *False*
show *?thesis* **by** (*rule sum.neutral*, *auto simp add*: *False b-not-j*)
next
case *True* — This case is different from the other cases
let *?f*=λk. (*if k = a then* (*0*::'*a*) + (*1*::'*a*) ∗ − *q else if b = k then 1*::'*a else* (*0*::'*a*)) ∗
(*if j = a then* (*if k = a then 1*::'*a else* (*0*::'*a*)) + (*if k = b then 1*::'*a else* (*0*::'*a*)) ∗ *q else if k = j then 1*::'*a else* (*0*::'*a*))
have *univ-eq*: *UNIV* = ((*UNIV* − {*a*}− {*b*}) ∪ ({*b*} ∪ {*a*})) **by** *auto*
have *sum-a*: *sum ?f* {*a*} = −*q* **unfolding** *True* **using** *b-not-j* **using** *a-noteq-b* **by** *auto*
have *sum-b*: *sum ?f* {*b*} = *q* **unfolding** *True* **using** *b-not-j* **using** *a-noteq-b* **by** *auto*
have *sum-rest*: *sum ?f* (*UNIV* − {*a*} − {*b*}) = *0* **by** (*rule sum.neutral*, *auto simp add*: *True b-not-j a-noteq-b*)
have *sum ?f UNIV* = *sum ?f* ((*UNIV* − {*a*}− {*b*}) ∪ ({*b*} ∪ {*a*})) **using** *univ-eq* **by** *simp*
also have ... = *sum ?f* (*UNIV* − {*a*} − {*b*}) + *sum ?f* ({*b*} ∪ {*a*}) **by** (*rule sum.union-disjoint*, *auto*)
also have ... = *sum ?f* (*UNIV* − {*a*} − {*b*}) + *sum ?f* {*b*} + *sum ?f* {*a*} **by** (*auto simp add*: *sum.union-disjoint a-noteq-b*)
also have ... = *0* **unfolding** *sum-a sum-b sum-rest* **by** *simp*
finally show *?thesis* **.**
qed
qed
qed

4.5 Relationships amongst the definitions

Relationships between *interchange-rows* and *interchange-columns*

lemma *interchange-rows-transpose*:
shows *interchange-rows* (*transpose A*) *a b* = *transpose* (*interchange-columns A a b*)
unfolding *interchange-rows-def interchange-columns-def transpose-def* **by** *vector*

lemma *interchange-rows-transpose′*:
shows *interchange-rows A a b* = *transpose* (*interchange-columns* (*transpose A*) *a b*)
unfolding *interchange-rows-def interchange-columns-def transpose-def* **by** *vector*

lemma *interchange-columns-transpose*:
shows *interchange-columns* (*transpose A*) *a b* = *transpose* (*interchange-rows A a b*)
unfolding *interchange-rows-def interchange-columns-def transpose-def* **by** *vector*

lemma *interchange-columns-transpose′*:
shows *interchange-columns A a b* = *transpose* (*interchange-rows* (*transpose A*) *a b*)
unfolding *interchange-rows-def interchange-columns-def transpose-def* **by** *vector*

4.6 Code Equations

Code equations for *interchange-rows ?A ?a ?b* = (*χi j. if i = ?a then ?A $?b $ j else if i = ?b then ?A $?a $ j else ?A $ i $ j*), *interchange-columns ?A ?n ?m* = (*χi j. if j = ?n then ?A $ i $?m else if j = ?m then ?A $ i $?n else ?A $ i $ j*), *row-add ?A ?a ?b ?q* = (*χi j. if i = ?a then ?A $?a $ j + ?q ∗ ?A $?b $ j else ?A $ i $ j*), *column-add ?A ?n ?m ?q* = (*χi j. if j = ?n then ?A $ i $?n + ?A $ i $?m ∗ ?q else ?A $ i $ j*), *mult-row ?A ?a ?q* = (*χi j. if i = ?a then ?q ∗ ?A $?a $ j else ?A $ i $ j*) and *mult-column ?A ?n ?q* = (*χi j. if j = ?n then ?A $ i $ j ∗ ?q else ?A $ i $ j*):

definition *interchange-rows-row*
where *interchange-rows-row A a b i* = *vec-lambda* (%*j. if i = a then A $ b $ j else if i = b then A $ a $ j else A $ i $ j*)

lemma *interchange-rows-code* [*code abstract*]:
vec-nth (*interchange-rows-row A a b i*) = (%*j. if i = a then A $ b $ j else if i = b then A $ a $ j else A $ i $ j*)
unfolding *interchange-rows-row-def* **by** *auto*

lemma *interchange-rows-code-nth* [*code abstract*]: *vec-nth* (*interchange-rows A a b*) = *interchange-rows-row A a b*
unfolding *interchange-rows-def* **unfolding** *interchange-rows-row-def*[*abs-def*]
by *auto*

definition *interchange-columns-row*
where *interchange-columns-row A n m i* = *vec-lambda* (%*j. if j = n then A $ i $ m else if j = m then A $ i $ n else A $ i $ j*)

lemma *interchange-columns-code* [*code abstract*]:
vec-nth (*interchange-columns-row A n m i*) = (%*j. if j = n then A $ i $ m else if j = m then A $ i $ n else A $ i $ j*)
unfolding *interchange-columns-row-def* **by** *auto*

lemma *interchange-columns-code-nth* [*code abstract*]: *vec-nth* (*interchange-columns A a b*) = *interchange-columns-row A a b*
unfolding *interchange-columns-def* **unfolding** *interchange-columns-row-def*[*abs-def*]
by *auto*

definition *row-add-row*
where *row-add-row A a b q i* = *vec-lambda* (%*j. if i = a then A $ a $ j + q ∗ A $ b $ j else A $ i $ j*)

lemma *row-add-code* [*code abstract*]:
vec-nth (*row-add-row A a b q i*) = (%*j. if i = a then A $ a $ j + q ∗ A $ b $ j else A $ i $ j*)
unfolding *row-add-row-def* **by** *auto*

lemma *row-add-code-nth* [*code abstract*]: *vec-nth* (*row-add A a b q*) = *row-add-row*

A a b q
unfolding *row-add-def* **unfolding** *row-add-row-def*[*abs-def*]
by *auto*

definition *column-add-row*
where *column-add-row A n m q i* = *vec-lambda* (%*j*. *if j* = *n then A* $ *i* $ *n* + *A* $ *i* $ *m* ∗ *q else A* $ *i* $ *j*)

lemma *column-add-code* [*code abstract*]:
vec-nth (*column-add-row A n m q i*) = (%*j*. *if j* = *n then A* $ *i* $ *n* + *A* $ *i* $ *m* ∗ *q else A* $ *i* $ *j*)
unfolding *column-add-row-def* **by** *auto*

lemma *column-add-code-nth* [*code abstract*]: *vec-nth* (*column-add A a b q*) = *column-add-row A a b q*
unfolding *column-add-def* **unfolding** *column-add-row-def*[*abs-def*]
by *auto*

definition *mult-row-row*
where *mult-row-row A a q i* = *vec-lambda* (%*j*. *if i* = *a then q* ∗ *A* $ *a* $ *j else A* $ *i* $ *j*)

lemma *mult-row-code* [*code abstract*]:
vec-nth (*mult-row-row A a q i*) = (%*j*. *if i* = *a then q* ∗ *A* $ *a* $ *j else A* $ *i* $ *j*)
unfolding *mult-row-row-def* **by** *auto*

lemma *mult-row-code-nth* [*code abstract*]: *vec-nth* (*mult-row A a q*) = *mult-row-row A a q*
unfolding *mult-row-def* **unfolding** *mult-row-row-def*[*abs-def*]
by *auto*

definition *mult-column-row*
where *mult-column-row A n q i* = *vec-lambda* (%*j*. *if j* = *n then A* $ *i* $ *j* ∗ *q else A* $ *i* $ *j*)

lemma *mult-column-code* [*code abstract*]:
vec-nth (*mult-column-row A n q i*) = (%*j*. *if j* = *n then A* $ *i* $ *j* ∗ *q else A* $ *i* $ *j*)
unfolding *mult-column-row-def* **by** *auto*

lemma *mult-column-code-nth* [*code abstract*]: *vec-nth* (*mult-column A a q*) = *mult-column-row A a q*
unfolding *mult-column-def* **unfolding** *mult-column-row-def*[*abs-def*]
by *auto*

end

5 Rank of a matrix

theory *Rank*
imports
Rank-Nullity-Theorem.Dim-Formula
begin

5.1 Row rank, column rank and rank

Definitions of row rank, column rank and rank

definition *row-rank* :: *'a*::{*field*}^'*n*^'*m*=>*nat*
where *row-rank A* = *vec.dim* (*row-space A*)

definition *col-rank* :: *'a*::{*field*}^'*n*^'*m*=>*nat*
where *col-rank A* = *vec.dim* (*col-space A*)

lemma *rank-def*: *rank A* = *row-rank A*
by (*auto simp*: *row-rank-def row-rank-def-gen row-space-def*)

5.2 Properties

lemma *rrk-is-preserved*:
fixes *A*::*'a*::{*field*}^'*cols*^'*rows*::{*finite*, *wellorder*}
and *P*::*'a*::{*field*}^'*rows*::{*finite*, *wellorder*}^'*rows*::{*finite*, *wellorder*}
assumes *inv-P*: *invertible P*
shows *row-rank A* = *row-rank* (*P*∗∗*A*)
by (*metis row-space-is-preserved row-rank-def inv-P*)

lemma *crk-is-preserved*:
fixes *A*::*'a*::{*field*}^'*cols*::{*finite*, *wellorder*}^'*rows*
and *P*::*'a*::{*field*}^'*rows*^'*rows*
assumes *inv-P*: *invertible P*
shows *col-rank A* = *col-rank* (*P*∗∗*A*)
using *rank-nullity-theorem-matrices* **unfolding** *ncols-def*
by (*metis col-rank-def inv-P add-left-cancel null-space-is-preserved*)

end

6 Gauss Jordan algorithm over abstract matrices

theory *Gauss-Jordan*
imports
Rref
Elementary-Operations
Rank
begin

6.1 The Gauss-Jordan Algorithm

Now, a computable version of the Gauss-Jordan algorithm is presented. The output will be a matrix in reduced row echelon form. We present an algorithm in which the reduction is applied by columns

Using this definition, zeros are made in the column j of a matrix A placing the pivot entry (a nonzero element) in the position (i,j). For that, a suitable row interchange is made to achieve a non-zero entry in position (i,j). Then, this pivot entry is multiplied by its inverse to make the pivot entry equals to 1. After that, are other entries of the j-th column are eliminated by subtracting suitable multiples of the i-th row from the other rows.

definition *Gauss-Jordan-in-ij* :: *'a*::{*semiring-1*, *inverse*, *one*, *uminus*}^'m^'n::{*finite*, *ord*}=> *'n*=>*'m*=>*'a*^'m^'n::{*finite*, *ord*}
where *Gauss-Jordan-in-ij A i j* = (*let n* = (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*);
interchange-A = (*interchange-rows A i n*);
A' = *mult-row interchange-A i* (*1*/*interchange-A*$*i*$*j*) *in*
vec-lambda(% *s. if s*=*i then A'* $ *s else* (*row-add A' s i* (−(*interchange-A*$*s*$*j*))) $ *s*))

lemma *Gauss-Jordan-in-ij-unfold*:
assumes ∃ *n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*
obtains *n* :: *'n*::{*finite*, *wellorder*} **and** *interchange-A* **and** *A'*
where
(*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) = *n*
and *A* $ *n* $ *j* ≠ *0*
and *i* ≤ *n*
and *interchange-A* = *interchange-rows A i n*
and *A'* = *mult-row interchange-A i* (*1* / *interchange-A* $ *i* $ *j*)
and *Gauss-Jordan-in-ij A i j* = *vec-lambda* (λ*s. if s* = *i then A'* $ *s*
else (*row-add A' s i* (− (*interchange-A* $ *s* $ *j*))) $ *s*)
proof −
from *assms* **obtain** *m* **where** *Anj*: *A* $ *m* $ *j* ≠ *0* ∧ *i* ≤ *m* **..**
moreover define *n* **where** *n* = (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*)
then have *P1*: (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) = *n* **by** *simp*
ultimately have *P2*: *A* $ *n* $ *j* ≠ *0* **and** *P3*: *i* ≤ *n*
using *LeastI* [*of* λ*n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n m*] **by** *simp-all*
define *interchange-A* **where** *interchange-A* = *interchange-rows A i n*
then have *P4*: *interchange-A* = *interchange-rows A i n* **by** *simp*
define *A'* **where** *A'* = *mult-row interchange-A i* (*1* / *interchange-A* $ *i* $ *j*)
then have *P5*: *A'* = *mult-row interchange-A i* (*1* / *interchange-A* $ *i* $ *j*) **by** *simp*
have *P6*: *Gauss-Jordan-in-ij A i j* = *vec-lambda* (λ*s. if s* = *i then A'* $ *s*
else (*row-add A' s i* (− (*interchange-A* $ *s* $ *j*))) $ *s*)
by (*simp only*: *Gauss-Jordan-in-ij-def P1 P4* [*symmetric*] *P5* [*symmetric*] *Let-def*)
from *P1 P2 P3 P4 P5 P6 that* **show** *thesis* **by** *blast*
qed

The following definition makes the step of Gauss-Jordan in a column. This function receives two input parameters: the column k where the step of Gauss-Jordan must be applied and a pair (which consists of the row where the pivot should be placed in the column k and the original matrix).

definition *Gauss-Jordan-column-k* :: (*nat* × ('*a*::{*zero*,*inverse*,*uminus*,*semiring-1*}^'*m*::{*mod-type*}^'*n*::{*mod-*

=> *nat* => (*nat* × ('*a*^'*m*::{*mod-type*}^'*n*::{*mod-type*}))
where *Gauss-Jordan-column-k A' k* = (*let i*=*fst A'*; *A*=(*snd A'*); *from-nat-i*=(*from-nat i*::'*n*); *from-nat-k*=(*from-nat k*::'*m*) *in*
if (∀ *m*≥(*from-nat-i*). *A* $ *m* $(*from-nat-k*)=*0*) ∨ (*i* = *nrows A*) *then* (*i*,*A*)
else (*i*+*1*, (*Gauss-Jordan-in-ij A* (*from-nat-i*) (*from-nat-k*))))

The following definition applies the Gauss-Jordan step from the first column up to the k one (included).

definition *Gauss-Jordan-upt-k* :: '*a*::{*inverse*,*uminus*,*semiring-1*}^'*columns*::{*mod-type*}^'*rows*::{*mod-type*}
=> *nat*
=> '*a*^'*columns*::{*mod-type*}^'*rows*::{*mod-type*}
where *Gauss-Jordan-upt-k A k* = *snd* (*foldl Gauss-Jordan-column-k* (*0*,*A*) [*0*..<*Suc k*])

Gauss-Jordan is to apply the *Gauss-Jordan-column-k* in all columns.

definition *Gauss-Jordan* :: '*a*::{*inverse*,*uminus*,*semiring-1*}^'*columns*::{*mod-type*}^'*rows*::{*mod-type*}

=> '*a*^'*columns*::{*mod-type*}^'*rows*::{*mod-type*}
where *Gauss-Jordan A* = *Gauss-Jordan-upt-k A* ((*ncols A*) − *1*)

6.2 Properties about rref and the greatest nonzero row.

lemma *greatest-plus-one-eq-0*:
fixes *A*::'*a*::{*field*}^'*columns*::{*mod-type*}^'*rows*::{*mod-type*} **and** *k*::*nat*
assumes *Suc* (*to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*)) = *nrows A*
shows (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1* = *0*
proof −
have *to-nat* (*GREATEST R*. ¬ *is-zero-row-upt-k R k A*) + *1* = *card* (*UNIV*::'*rows set*)
using *assms* **unfolding** *nrows-def* **by** *fastforce*
thus (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + (*1*::'*rows*) = (*0*::'*rows*)
using *to-nat-plus-one-less-card* **by** *fastforce*
qed

lemma *from-nat-to-nat-greatest*:
fixes *A*::'*a*::{*zero*}^'*columns*::{*mod-type*}^'*rows*::{*mod-type*}
shows *from-nat* (*Suc* (*to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*))) = (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1*
unfolding *Suc-eq-plus1*
unfolding *to-nat-1*[**where** *?'a*='*rows*, *symmetric*]
unfolding *add-to-nat-def* **..**

lemma *greatest-less-zero-row*:
fixes *A*::*'a*::{*one*, *zero*}^*'n*::{*mod-type*}^*'m*::{*finite*,*one*,*plus*,*linorder*}
assumes *r*: *reduced-row-echelon-form-upt-k A k*
and *zero-i*: *is-zero-row-upt-k i k A*
and *not-all-zero*: ¬ (∀ *a*. *is-zero-row-upt-k a k A*)
shows (*GREATEST m*. ¬ *is-zero-row-upt-k m k A*) < *i*
proof (*rule ccontr*)
assume *not-less-i*: ¬ (*GREATEST m*. ¬ *is-zero-row-upt-k m k A*) < *i*
have *i-less-greatest*: *i* < (*GREATEST m*. ¬ *is-zero-row-upt-k m k A*)
by (*metis* (*mono-tags*, *lifting*) *GreatestI neqE not-all-zero not-less-i zero-i*)
have *is-zero-row-upt-k* (*GREATEST m*. ¬ *is-zero-row-upt-k m k A*) *k A*
using *r zero-i i-less-greatest* **unfolding** *reduced-row-echelon-form-upt-k-def* **by** *blast*
thus *False* **using** *GreatestI-ex not-all-zero* **by** *fast*
qed

lemma *rref-suc-if-zero-below-greatest*:
fixes *A*::*'a*::{*one*, *zero*}^*'n*::{*mod-type*}^*'m*::{*finite*,*one*,*plus*,*linorder*}
assumes *r*: *reduced-row-echelon-form-upt-k A k*
and *not-all-zero*: ¬ (∀ *a*. *is-zero-row-upt-k a k A*)
and *all-zero-below-greatest*: ∀ *a*. *a*>(*GREATEST m*. ¬ *is-zero-row-upt-k m k A*) ⟶ *is-zero-row-upt-k a* (*Suc k*) *A*
shows *reduced-row-echelon-form-upt-k A* (*Suc k*)
proof (*rule reduced-row-echelon-form-upt-k-intro*, *auto*)
fix *i j* **assume** *zero-i-suc*: *is-zero-row-upt-k i* (*Suc k*) *A* **and** *i-le-j*: *i* < *j*
have *zero-i*: *is-zero-row-upt-k i k A* **using** *zero-i-suc* **unfolding** *is-zero-row-upt-k-def* **by** *simp*
have *i*>(*GREATEST m*. ¬ *is-zero-row-upt-k m k A*) **by** (*rule greatest-less-zero-row*[*OF r zero-i not-all-zero*])
hence *j*>(*GREATEST m*. ¬ *is-zero-row-upt-k m k A*) **using** *i-le-j* **by** *simp*
thus *is-zero-row-upt-k j* (*Suc k*) *A* **using** *all-zero-below-greatest* **by** *fast*
next
fix *i* **assume** *not-zero-i*: ¬ *is-zero-row-upt-k i* (*Suc k*) *A*
show *A* $ *i* $ (*LEAST k*. *A* $ *i* $ *k* ≠ *0*) = *1*
using *greatest-less-zero-row*[*OF r - not-all-zero*] *not-zero-i r all-zero-below-greatest*
unfolding *reduced-row-echelon-form-upt-k-def*
by *fast*
next
fix *i*
assume *i*: *i* < *i* + *1* **and** *not-zero-i*: ¬ *is-zero-row-upt-k i* (*Suc k*) *A* **and** *not-zero-suc-i*: ¬ *is-zero-row-upt-k* (*i* + *1*) (*Suc k*) *A*
have *not-zero-i-k*: ¬ *is-zero-row-upt-k i k A*
using *all-zero-below-greatest greatest-less-zero-row*[*OF r - not-all-zero*] *not-zero-i* **by** *blast*
have *not-zero-suc-i*: ¬ *is-zero-row-upt-k* (*i*+*1*) *k A*
using *all-zero-below-greatest greatest-less-zero-row*[*OF r - not-all-zero*] *not-zero-suc-i* **by** *blast*
have *aux*:(∀ *i j*. *i* + *1* = *j* ∧ *i* < *j* ∧ ¬ *is-zero-row-upt-k i k A* ∧ ¬ *is-zero-row-upt-k j k A* ⟶ (*LEAST n*. *A* $ *i* $ *n* ≠ *0*) < (*LEAST n*. *A* $ *j* $ *n* ≠ *0*))

using r **unfolding** *reduced-row-echelon-form-upt-k-def* **by** *fast*
show $(LEAST\ n.\ A\ \$\ i\ \$\ n \neq 0) < (LEAST\ n.\ A\ \$\ (i+1)\ \$\ n \neq 0)$ **using** *aux not-zero-i-k not-zero-suc-i i* **by** *simp*
next
fix $i\ j$ **assume** $\neg$ *is-zero-row-upt-k i* (*Suc k*) A **and** $i \neq j$
thus $A\ \$\ j\ \$\ (LEAST\ n.\ A\ \$\ i\ \$\ n \neq 0) = 0$
using *all-zero-below-greatest greatest-less-zero-row not-all-zero r rref-upt-condition4* **by** *blast*
qed

lemma *rref-suc-if-all-rows-not-zero*:
fixes A::'a::{*one*, *zero*}^'n::{*mod-type*}^'m::{*finite*,*one*,*plus*,*linorder*}
assumes r: *reduced-row-echelon-form-upt-k A k*
and *all-not-zero*: $\forall\, n.\ \neg$ *is-zero-row-upt-k n k A*
shows *reduced-row-echelon-form-upt-k A* (*Suc k*)
proof (*rule rref-suc-if-zero-below-greatest*)
show *reduced-row-echelon-form-upt-k A k* **using** r **.**
show $\neg\ (\forall\, a.$ *is-zero-row-upt-k a k A*) **using** *all-not-zero* **by** *auto*
show $\forall\, a{>}GREATEST\ m.\ \neg$ *is-zero-row-upt-k m k A. is-zero-row-upt-k a* (*Suc k*) *A*
using *all-not-zero not-greater-Greatest* **by** *blast*
qed

lemma *greatest-ge-nonzero-row*:
fixes A::'a::{*zero*}^'n::{*mod-type*}^'m::{*finite*,*linorder*}
assumes $\neg$ *is-zero-row-upt-k i k A*
shows $i \leq (GREATEST\ m.\ \neg$ *is-zero-row-upt-k m k A*) **using** *Greatest-ge*[*of* ($\lambda m.\ \neg$ *is-zero-row-upt-k m k A*), *OF assms*] **.**

lemma *greatest-ge-nonzero-row'*:
fixes A::'a::{*zero*, *one*}^'n::{*mod-type*}^'m::{*finite*, *linorder*, *one*, *plus*}
assumes r: *reduced-row-echelon-form-upt-k A k*
and i: $i \leq (GREATEST\ m.\ \neg$ *is-zero-row-upt-k m k A*)
and *not-all-zero*: $\neg\ (\forall\, a.$ *is-zero-row-upt-k a k A*)
shows $\neg$ *is-zero-row-upt-k i k A*
using *greatest-less-zero-row*[*OF r*] *i not-all-zero* **by** *fastforce*

corollary *row-greater-greatest-is-zero*:
fixes A::'a::{*zero*}^'n::{*mod-type*}^'m::{*finite*,*linorder*}
assumes $(GREATEST\ m.\ \neg$ *is-zero-row-upt-k m k A*$) < i$
shows *is-zero-row-upt-k i k A* **using** *greatest-ge-nonzero-row assms* **by** *fastforce*

6.3 The proof of its correctness

Properties of *Gauss-Jordan-in-ij*

lemma *Gauss-Jordan-in-ij-1*:
fixes A::'a::{*field*}^'m^'n::{*finite*, *ord*, *wellorder*}
assumes *ex*: $\exists\, n.\ A\ \$\ n\ \$\ j \neq 0 \wedge i \leq n$

shows (*Gauss-Jordan-in-ij A i j*) $ *i* $ *j* = *1*
proof (*unfold Gauss-Jordan-in-ij-def Let-def mult-row-def interchange-rows-def*, *vector*)
obtain *n* **where** *Anj*: *A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n* **using** *ex* **by** *blast*
show *A* $ (*LEAST n*. *A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) $ *j* ≠ *0* **using** *LeastI*[*of* λ*n*. *A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n n*, *OF Anj*] **by** *simp*
qed

lemma *Gauss-Jordan-in-ij-0*:
fixes *A*::*'a*::{*field*}^*'m*^*n*::{*finite*, *ord*, *wellorder*}
assumes *ex*: ∃ *n*. *A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n* **and** *a*: *a* ≠ *i*
shows (*Gauss-Jordan-in-ij A i j*) $ *a* $ *j* = *0*
using *ex* **apply** (*rule Gauss-Jordan-in-ij-unfold*) **using** *a* **by** (*simp add*: *mult-row-def interchange-rows-def row-add-def*)

corollary *Gauss-Jordan-in-ij-0'*:
fixes *A*::*'a*::{*field*}^*'m*^*n*::{*finite*, *ord*, *wellorder*}
assumes *ex*: ∃ *n*. *A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*
shows ∀ *a*. *a* ≠ *i* ⟶ (*Gauss-Jordan-in-ij A i j*) $ *a* $ *j* = *0* **using** *assms Gauss-Jordan-in-ij-0* **by** *blast*

lemma *Gauss-Jordan-in-ij-preserves-previous-elements*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *r*: *reduced-row-echelon-form-upt-k A k*
and *not-zero-a*: ¬ *is-zero-row-upt-k a k A*
and *exists-m*: ∃ *m*. *A* $ *m* $ (*from-nat k*) ≠ *0* ∧ (*GREATEST m*. ¬ *is-zero-row-upt-k m k A*) + *1* ≤ *m*
and *Greatest-plus-1*: (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1* ≠ *0*
and *j-le-k*: *to-nat j* < *k*
shows *Gauss-Jordan-in-ij A* ((*GREATEST m*. ¬ *is-zero-row-upt-k m k A*) + *1*) (*from-nat k*) $ *i* $ *j* = *A* $ *i* $ *j*
proof (*unfold Gauss-Jordan-in-ij-def Let-def interchange-rows-def mult-row-def row-add-def*, *auto*)
define *last-nonzero-row* **where** *last-nonzero-row* = (*GREATEST m*. ¬ *is-zero-row-upt-k m k A*)
have *last-nonzero-row* < (*last-nonzero-row* + *1*) **by** (*rule Suc-le'*[*of last-nonzero-row*], *auto simp add*: *last-nonzero-row-def Greatest-plus-1*)
hence *zero-row*: *is-zero-row-upt-k* (*last-nonzero-row* + *1*) *k A*
using *not-le greatest-ge-nonzero-row last-nonzero-row-def* **by** *fastforce*
hence *A-greatest-0*: *A* $ (*last-nonzero-row* + *1*) $ *j* = *0* **unfolding** *is-zero-row-upt-k-def last-nonzero-row-def* **using** *j-le-k* **by** *auto*
then show *A* $ (*last-nonzero-row* + *1*) $ *j* / *A* $ (*last-nonzero-row* + *1*) $ *from-nat k* = *A* $ (*last-nonzero-row* + *1*) $ *j*
by *simp*
show *zero*: *A* $ (*LEAST n*. *A* $ *n* $ *from-nat k* ≠ *0* ∧ (*GREATEST m*. ¬ *is-zero-row-upt-k m k A*) + *1* ≤ *n*) $ *j* = *0*
proof −
define *least-n* **where** *least-n* = (*LEAST n*. *A* $ *n* $ *from-nat k* ≠ *0* ∧ (*GREATEST m*. ¬ *is-zero-row-upt-k m k A*) + *1* ≤ *n*)

have $\exists n.\ A\ \$\ n\ \$\ from\text{-}nat\ k \neq 0 \wedge (GREATEST\ m.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ m\ k\ A) + 1 \leq n$ **by** (*metis exists-m*)
from *this* **obtain** *n* **where** *n1*: $A\ \$\ n\ \$\ from\text{-}nat\ k \neq 0$ **and** *n2*: $(GREATEST\ m.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ m\ k\ A) + 1 \leq n$ **by** *blast*
have $(GREATEST\ m.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ m\ k\ A) + 1 \leq least\text{-}n$
by (*metis* (*lifting*, *full-types*) *LeastI-ex least-n-def n1 n2*)
hence *is-zero-row-upt-k least-n k A* **using** *last-nonzero-row-def less-le rref-upt-condition1*[*OF r*] *zero-row* **by** *metis*
thus $A\ \$\ least\text{-}n\ \$\ j = 0$ **unfolding** *is-zero-row-upt-k-def* **using** *j-le-k* **by** *simp*
qed
show $A\ \$\ ((GREATEST\ m.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ m\ k\ A) + 1)\ \$\ j -$
$A\ \$\ ((GREATEST\ m.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ m\ k\ A) + 1)\ \$\ from\text{-}nat\ k *$
$A\ \$\ (LEAST\ n.\ A\ \$\ n\ \$\ from\text{-}nat\ k \neq 0 \wedge (GREATEST\ m.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ m\ k\ A) + 1 \leq n)\ \$\ j\ /$
$A\ \$\ (LEAST\ n.\ A\ \$\ n\ \$\ from\text{-}nat\ k \neq 0 \wedge (GREATEST\ m.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ m\ k\ A) + 1 \leq n)\ \$\ from\text{-}nat\ k =$
$A\ \$\ (LEAST\ n.\ A\ \$\ n\ \$\ from\text{-}nat\ k \neq 0 \wedge (GREATEST\ m.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ m\ k\ A) + 1 \leq n)\ \$\ j$
unfolding *last-nonzero-row-def*[*symmetric*] **unfolding** *A-greatest-0* **unfolding** *last-nonzero-row-def* **unfolding** *zero* **by** *fastforce*
show $A\ \$\ (LEAST\ n.\ A\ \$\ n\ \$\ from\text{-}nat\ k \neq 0 \wedge (GREATEST\ m.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ m\ k\ A) + 1 \leq n)\ \$\ j\ /$
$A\ \$\ (LEAST\ n.\ A\ \$\ n\ \$\ from\text{-}nat\ k \neq 0 \wedge (GREATEST\ m.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ m\ k\ A) + 1 \leq n)\ \$\ from\text{-}nat\ k =$
$A\ \$\ ((GREATEST\ m.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ m\ k\ A) + 1)\ \$\ j$ **unfolding** *zero* **using** *A-greatest-0* **unfolding** *last-nonzero-row-def* **by** *simp*
qed

lemma *Gauss-Jordan-in-ij-preserves-previous-elements'*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *all-zero*: $\forall n.\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ n\ k\ A$
and *j-le-k*: $to\text{-}nat\ j < k$
and *A-nk-not-zero*: $A\ \$\ n\ \$\ (from\text{-}nat\ k) \neq 0$
shows $Gauss\text{-}Jordan\text{-}in\text{-}ij\ A\ 0\ (from\text{-}nat\ k)\ \$\ i\ \$\ j = A\ \$\ i\ \$\ j$
proof (*unfold Gauss-Jordan-in-ij-def Let-def mult-row-def interchange-rows-def row-add-def*, *auto*)
have *A-0-j*: $A\ \$\ 0\ \$\ j = 0$ **using** *all-zero is-zero-row-upt-k-def j-le-k* **by** *blast*
then show $A\ \$\ 0\ \$\ j\ /\ A\ \$\ 0\ \$\ from\text{-}nat\ k = A\ \$\ 0\ \$\ j$ **by** *simp*
show *A-least-j*: $A\ \$\ (LEAST\ n.\ A\ \$\ n\ \$\ from\text{-}nat\ k \neq 0 \wedge 0 \leq n)\ \$\ j = 0$ **using** *all-zero is-zero-row-upt-k-def j-le-k* **by** *blast*
show $A\ \$\ 0\ \$\ j -$
$A\ \$\ 0\ \$\ from\text{-}nat\ k * A\ \$\ (LEAST\ n.\ A\ \$\ n\ \$\ from\text{-}nat\ k \neq 0 \wedge 0 \leq n)\ \$\ j\ /$
$A\ \$\ (LEAST\ n.\ A\ \$\ n\ \$\ from\text{-}nat\ k \neq 0 \wedge 0 \leq n)\ \$\ from\text{-}nat\ k =$
$A\ \$\ (LEAST\ n.\ A\ \$\ n\ \$\ from\text{-}nat\ k \neq 0 \wedge 0 \leq n)\ \$\ j$ **unfolding** *A-0-j A-least-j* **by** *fastforce*
show $A\ \$\ (LEAST\ n.\ A\ \$\ n\ \$\ from\text{-}nat\ k \neq 0 \wedge 0 \leq n)\ \$\ j\ /\ A\ \$\ (LEAST\ n.\ A\ \$\ n\ \$\ from\text{-}nat\ k \neq 0 \wedge 0 \leq n)\ \$\ from\text{-}nat\ k = A\ \$\ 0\ \$\ j$

unfolding *A-least-j A-0-j* **by** *simp*
qed

lemma *is-zero-after-Gauss*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'m*::{*mod-type*}
assumes *zero-a*: *is-zero-row-upt-k a k A*
and *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
and *r*: *reduced-row-echelon-form-upt-k A k*
and *greatest-less-ma*: (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1* ≤ *ma*
and *A-ma-k-not-zero*: *A* $ *ma* $ *from-nat k* ≠ *0*
shows *is-zero-row-upt-k a k* (*Gauss-Jordan-in-ij A* ((*GREATEST m*. ¬ *is-zero-row-upt-k m k A*) + *1*) (*from-nat k*))
proof (*subst is-zero-row-upt-k-def*, *clarify*)
fix *j*::*'n* **assume** *j-less-k*: *to-nat j* < *k*
have *not-zero-g*: (*GREATEST m*. ¬ *is-zero-row-upt-k m k A*) + *1* ≠ *0*
proof (*rule ccontr*, *simp*)
assume (*GREATEST m*. ¬ *is-zero-row-upt-k m k A*) + *1* = *0*
hence (*GREATEST m*. ¬ *is-zero-row-upt-k m k A*) = −*1* **using** *a-eq-minus-1*
by *blast*
hence *a*≤(*GREATEST m*. ¬ *is-zero-row-upt-k m k A*) **using** *Greatest-is-minus-1*
by *auto*
hence ¬ *is-zero-row-upt-k a k A* **using** *greatest-less-zero-row*[*OF r*] *not-zero-m*
by *fastforce*
thus *False* **using** *zero-a* **by** *contradiction*
qed
have *Gauss-Jordan-in-ij A* ((*GREATEST m*. ¬ *is-zero-row-upt-k m k A*) + *1*) (*from-nat k*) $ *a* $ *j* = *A* $ *a* $ *j*
by (*rule Gauss-Jordan-in-ij-preserves-previous-elements*[*OF r not-zero-m - not-zero-g j-less-k*], *auto intro*!: *A-ma-k-not-zero greatest-less-ma*)
also have ... = *0*
using *zero-a j-less-k* **unfolding** *is-zero-row-upt-k-def* **by** *blast*
finally show *Gauss-Jordan-in-ij A* ((*GREATEST m*. ¬ *is-zero-row-upt-k m k A*) + *1*) (*from-nat k*) $ *a* $ *j* = *0* **.**
qed

lemma *all-zero-imp-Gauss-Jordan-column-not-zero-in-row-0*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*if* ∀ *m*. *is-zero-row-upt-k m k A then 0 else to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1*)
defines *B*:*B*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *all-zero*: ∀ *n*. *is-zero-row-upt-k n k A*
and *not-zero-i*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
and *Amk-zero*: *A* $ *m* $ *from-nat k* ≠ *0*
shows *i*=*0*
proof (*rule ccontr*)
assume *i-not-0*: *i* ≠ *0*
have *ia2*: *ia* = *0* **using** *ia all-zero* **by** *simp*
have *B-eq-Gauss*: *B* = *Gauss-Jordan-in-ij A 0* (*from-nat k*)

unfolding *B Gauss-Jordan-column-k-def Let-def fst-conv snd-conv ia2*
using *all-zero Amk-zero least-mod-type* **unfolding** *from-nat-0 nrows-def* **by** *auto*
also have ...$ *i* $ (*from-nat k*) = *0* **proof** (*rule Gauss-Jordan-in-ij-0*)
show ∃ *n*. *A* $ *n* $ *from-nat k* ≠ *0* ∧ *0* ≤ *n* **using** *Amk-zero least-mod-type* **by** *blast*
show *i* ≠ *0* **using** *i-not-0* **.**
qed
finally have *B* $ *i* $ *from-nat k* = *0* **.**
hence *is-zero-row-upt-k i* (*Suc k*) *B*
unfolding *B-eq-Gauss*
using *Gauss-Jordan-in-ij-preserves-previous-elements'*[*OF all-zero - Amk-zero*]
by (*metis all-zero is-zero-row-upt-k-def less-SucE to-nat-from-nat*)
thus *False* **using** *not-zero-i* **by** *contradiction*
qed

Here we start to prove that the output of *Gauss Jordan A* is a matrix in reduced row echelon form.

lemma *condition-1-part-1*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
assumes *zero-column-k*: ∀ *m*≥*from-nat 0*. *A* $ *m* $ *from-nat k* = *0*
and *all-zero*: ∀ *m*. *is-zero-row-upt-k m k A*
shows *is-zero-row-upt-k j* (*Suc k*) *A*
unfolding *is-zero-row-upt-k-def* **apply** *clarify*
proof −
fix *ja*::*'columns* **assume** *ja-less-suc-k*: *to-nat ja* < *Suc k*
show *A* $ *j* $ *ja* = *0*
proof (*cases to-nat ja* < *k*)
case *True* **thus** *?thesis* **using** *all-zero* **unfolding** *is-zero-row-upt-k-def* **by** *blast*
next
case *False* **hence** *ja-eq-k*: *k* = *to-nat ja* **using** *ja-less-suc-k* **by** *simp*
show *?thesis* **using** *zero-column-k* **unfolding** *ja-eq-k from-nat-to-nat-id from-nat-0*
using *least-mod-type* **by** *blast*
qed
qed

lemma *condition-1-part-2*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
assumes *j-not-zero*: *j* ≠ *0*
and *all-zero*: ∀ *m*. *is-zero-row-upt-k m k A*
and *Amk-not-zero*: *A* $ *m* $ *from-nat k* ≠ *0*
shows *is-zero-row-upt-k j* (*Suc k*) (*Gauss-Jordan-in-ij A* (*from-nat 0*) (*from-nat k*))
proof (*unfold is-zero-row-upt-k-def*, *clarify*)
fix *ja*::*'columns*
assume *ja-less-suc-k*: *to-nat ja* < *Suc k*
show *Gauss-Jordan-in-ij A* (*from-nat 0*) (*from-nat k*) $ *j* $ *ja* = *0*
proof (*cases to-nat ja* < *k*)
case *True*

```
    have Gauss-Jordan-in-ij A (from-nat 0) (from-nat k) $ j $ ja = A $ j $ ja
    unfolding from-nat-0 using Gauss-Jordan-in-ij-preserves-previous-elements'[OF
all-zero True Amk-not-zero] .
   also have ... = 0 using all-zero True unfolding is-zero-row-upt-k-def by blast
    finally show ?thesis .
  next
    case False hence k-eq-ja: k = to-nat ja
      using ja-less-suc-k by simp
    show Gauss-Jordan-in-ij A (from-nat 0) (from-nat k) $ j $ ja = 0
      unfolding k-eq-ja from-nat-to-nat-id
    proof (rule Gauss-Jordan-in-ij-0)
      show ∃ n. A $ n $ ja ≠ 0 ∧ from-nat 0 ≤ n
        using least-mod-type Amk-not-zero
        unfolding k-eq-ja from-nat-to-nat-id from-nat-0 by blast
      show j ≠ from-nat 0 using j-not-zero unfolding from-nat-0 .
    qed
  qed
qed

lemma condition-1-part-3:
  fixes A::'a::{field}^'columns::{mod-type}^'rows::{mod-type} and k::nat
  defines ia:ia≡(if ∀ m. is-zero-row-upt-k m k A then 0 else to-nat (GREATEST
n. ¬ is-zero-row-upt-k n k A) + 1)
  defines B:B ≡(snd (Gauss-Jordan-column-k (ia,A) k))
  assumes rref: reduced-row-echelon-form-upt-k A k
  and i-less-j: i<j
  and not-zero-m: ¬ is-zero-row-upt-k m k A
  and zero-below-greatest: ∀ m≥(GREATEST n. ¬ is-zero-row-upt-k n k A) + 1.
A $ m $ from-nat k = 0
  and zero-i-suc-k: is-zero-row-upt-k i (Suc k) B
  shows is-zero-row-upt-k j (Suc k) A
proof (unfold is-zero-row-upt-k-def, auto)
  fix ja::'columns
  assume ja-less-suc-k: to-nat ja < Suc k
  have ia2: ia=to-nat (GREATEST n. ¬ is-zero-row-upt-k n k A) + 1 unfolding
ia using not-zero-m by presburger
  have B-eq-A: B=A
    unfolding B Gauss-Jordan-column-k-def Let-def fst-conv snd-conv ia2
    apply simp
    unfolding from-nat-to-nat-greatest using zero-below-greatest by blast
   have zero-ikA: is-zero-row-upt-k i k A using zero-i-suc-k unfolding B-eq-A
is-zero-row-upt-k-def by fastforce
  hence zero-jkA: is-zero-row-upt-k j k A using rref-upt-condition1[OF rref] i-less-j
by blast
  show A $ j $ ja = 0
  proof (cases to-nat ja < k)
    case True
    thus ?thesis using zero-jkA unfolding is-zero-row-upt-k-def by blast
  next
```

case *False*
hence *k-eq-ja*:$k = to\text{-}nat\ ja$ **using** *ja-less-suc-k* **by** *auto*
have $(GREATEST\ n.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ n\ k\ A) + 1 \leq j$
proof (*rule le-Suc*, *rule GreatestI2*)
show $\neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ m\ k\ A$ **using** *not-zero-m* **.**
fix x **assume** *not-zero-xkA*: $\neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ x\ k\ A$ **show** $x < j$
using *rref-upt-condition1*[*OF rref*] *not-zero-xkA zero-jkA neq-iff* **by** *blast*
qed
thus *?thesis* **using** *zero-below-greatest* **unfolding** *k-eq-ja from-nat-to-nat-id is-zero-row-upt-k-def* **by** *blast*
qed
qed

lemma *condition-1-part-4*:
fixes A::$'a$::{*field*}^$'$*columns*::{*mod-type*}^$'$*rows*::{*mod-type*} **and** k::*nat*
defines *ia*:$ia \equiv (if\ \forall\ m.\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ m\ k\ A\ then\ 0\ else\ to\text{-}nat\ (GREATEST\ n.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ n\ k\ A) + 1)$
defines B:$B \equiv (snd\ (Gauss\text{-}Jordan\text{-}column\text{-}k\ (ia,A)\ k))$
assumes *rref*: *reduced-row-echelon-form-upt-k A k*
and *zero-i-suc-k*: *is-zero-row-upt-k i* (*Suc k*) *B*
and *i-less-j*: $i<j$
and *not-zero-m*: $\neg$ *is-zero-row-upt-k m k A*
and *greatest-eq-card*: $Suc\ (to\text{-}nat\ (GREATEST\ n.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ n\ k\ A)) = nrows\ A$
shows *is-zero-row-upt-k j* (*Suc k*) *A*
proof −
have *ia2*: $ia = to\text{-}nat\ (GREATEST\ n.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ n\ k\ A) + 1$ **unfolding** *ia* **using** *not-zero-m* **by** *presburger*
have *B-eq-A*: $B=A$
unfolding *B Gauss-Jordan-column-k-def Let-def fst-conv snd-conv ia2*
unfolding *from-nat-to-nat-greatest* **using** *greatest-eq-card nrows-def* **by** *force*
have *rref-Suc*: *reduced-row-echelon-form-upt-k A* (*Suc k*)
proof (*rule rref-suc-if-zero-below-greatest*[*OF rref*])
show $\forall\ a > GREATEST\ m.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ m\ k\ A.\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ a\ (Suc\ k)\ A$
using *greatest-eq-card not-less-eq to-nat-less-card to-nat-mono nrows-def* **by** *metis*
show $\neg\ (\forall\ a.\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ a\ k\ A)$ **using** *not-zero-m* **by** *fast*
qed
show *?thesis* **using** *zero-i-suc-k* **unfolding** *B-eq-A* **using** *rref-upt-condition1*[*OF rref-Suc*] *i-less-j* **by** *fast*
qed

lemma *condition-1-part-5*:
fixes A::$'a$::{*field*}^$'$*columns*::{*mod-type*}^$'$*rows*::{*mod-type*} **and** k::*nat*
defines *ia*:$ia \equiv (if\ \forall\ m.\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ m\ k\ A\ then\ 0\ else\ to\text{-}nat\ (GREATEST\ n.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ n\ k\ A) + 1)$
defines B:$B \equiv (snd\ (Gauss\text{-}Jordan\text{-}column\text{-}k\ (ia,A)\ k))$

assumes *rref*: *reduced-row-echelon-form-upt-k A k*
and *zero-i-suc-k*: *is-zero-row-upt-k i* (*Suc k*) *B*
and *i-less-j*: *i<j*
and *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
and *greatest-not-card*: *Suc* (*to-nat* (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*)) ≠ *nrows A*
and *greatest-less-ma*: (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1* ≤ *ma*
and *A-ma-k-not-zero*: *A $ ma $ from-nat k* ≠ *0*
shows *is-zero-row-upt-k j* (*Suc k*) (*Gauss-Jordan-in-ij A* ((*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*))
proof (*subst* (*1*) *is-zero-row-upt-k-def*, *clarify*)
fix *ja*::*'columns* **assume** *ja-less-suc-k*: *to-nat ja* < *Suc k*
have *ia2*: *ia=to-nat* (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1* **unfolding** *ia* **using** *not-zero-m* **by** *presburger*
have *B-eq-Gauss-ij*: *B* = *Gauss-Jordan-in-ij A* ((*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*)
unfolding *B Gauss-Jordan-column-k-def*
unfolding *ia2 Let-def fst-conv snd-conv*
using *greatest-not-card greatest-less-ma A-ma-k-not-zero*
by (*auto simp add*: *from-nat-to-nat-greatest nrows-def*)
have *zero-ikA*: *is-zero-row-upt-k i k A*
proof (*unfold is-zero-row-upt-k-def*, *clarify*)
fix *a*::*'columns*
assume *a-less-k*: *to-nat a* < *k*
have *A $ i $ a* = *Gauss-Jordan-in-ij A* ((*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) *$ i $ a*
proof (*rule Gauss-Jordan-in-ij-preserves-previous-elements*[*symmetric*])
show *reduced-row-echelon-form-upt-k A k* **using** *rref* **.**
show ¬ *is-zero-row-upt-k m k A* **using** *not-zero-m* **.**
show ∃ *n. A $ n $ from-nat k* ≠ *0* ∧ (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1* ≤ *n* **using** *A-ma-k-not-zero greatest-less-ma* **by** *blast*
show (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1* ≠ *0* **using** *suc-not-zero greatest-not-card* **unfolding** *nrows-def* **by** *simp*
show *to-nat a* < *k* **using** *a-less-k* **.**
qed
also have *... = 0* **unfolding** *B-eq-Gauss-ij*[*symmetric*] **using** *zero-i-suc-k a-less-k* **unfolding** *is-zero-row-upt-k-def* **by** *simp*
finally show *A $ i $ a = 0* **.**
qed
hence *zero-jkA*: *is-zero-row-upt-k j k A* **using** *rref-upt-condition1*[*OF rref*] *i-less-j* **by** *blast*
show *Gauss-Jordan-in-ij A* ((*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) *$ j $ ja = 0*
proof (*cases to-nat ja* < *k*)
case *True*
have *Gauss-Jordan-in-ij A* ((*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) *$ j $ ja = A $ j $ ja*
proof (*rule Gauss-Jordan-in-ij-preserves-previous-elements*)
show *reduced-row-echelon-form-upt-k A k* **using** *rref* **.**

show ¬ *is-zero-row-upt-k m k A* **using** *not-zero-m* **.**
show ∃ *n*. *A* $ *n* $ *from-nat k* ≠ *0* ∧ (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1* ≤ *n* **using** *A-ma-k-not-zero greatest-less-ma* **by** *blast*
show (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1* ≠ *0* **using** *suc-not-zero greatest-not-card* **unfolding** *nrows-def* **by** *simp*
show *to-nat ja* < *k* **using** *True* **.**
qed
also have ... = *0* **using** *zero-jkA True* **unfolding** *is-zero-row-upt-k-def* **by** *fast*
finally show *?thesis* **.**
next
case *False* **hence** *k-eq-ja*: *k* = *to-nat ja* **using** *ja-less-suc-k* **by** *simp*
show *?thesis*
proof (*unfold k-eq-ja from-nat-to-nat-id*, *rule Gauss-Jordan-in-ij-0*)
show ∃ *n*. *A* $ *n* $ *ja* ≠ *0* ∧ (*GREATEST n*. ¬ *is-zero-row-upt-k n* (*to-nat ja*) *A*) + *1* ≤ *n*
using *A-ma-k-not-zero greatest-less-ma k-eq-ja to-nat-from-nat* **by** *auto*
show *j* ≠ (*GREATEST n*. ¬ *is-zero-row-upt-k n* (*to-nat ja*) *A*) + *1*
proof (*unfold k-eq-ja*[*symmetric*], *rule ccontr*)
assume ¬ *j* ≠ (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1*
hence *j-eq*: *j* = (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1* **by** *fast*
hence *i* < (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1* **using** *i-less-j* **by** *force*
hence *i-le-greatest*: *i* ≤ (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) **using** *le-Suc not-less* **by** *auto*
hence ¬ *is-zero-row-upt-k i k A* **using** *greatest-ge-nonzero-row'*[*OF rref*] *not-zero-m* **by** *fast*
thus *False* **using** *zero-ikA* **by** *contradiction*
qed
qed
qed
qed

lemma *condition-1*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*if* ∀ *m*. *is-zero-row-upt-k m k A then 0 else to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1*)
defines *B*:*B*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *rref*: *reduced-row-echelon-form-upt-k A k*
and *zero-i-suc-k*: *is-zero-row-upt-k i* (*Suc k*) *B* **and** *i-less-j*: *i* < *j*
shows *is-zero-row-upt-k j* (*Suc k*) *B*
proof (*unfold B Gauss-Jordan-column-k-def ia Let-def fst-conv snd-conv*, *auto*, *unfold from-nat-to-nat-greatest*)
assume *zero-k*: ∀ *m*≥*from-nat 0*. *A* $ *m* $ *from-nat k* = *0* **and** *all-zero*: ∀ *m*. *is-zero-row-upt-k m k A*
show *is-zero-row-upt-k j* (*Suc k*) *A*
using *condition-1-part-1*[*OF zero-k all-zero*] **.**
next
fix *m*

assume *all-zero*: $\forall m.$ *is-zero-row-upt-k m k A* **and** *Amk-not-zero*: *A $ m $ from-nat k* $\neq$ *0*
have *j-not-0*: *j* $\neq$ *0* **using** *i-less-j least-mod-type not-le* **by** *blast*
show *is-zero-row-upt-k j* (*Suc k*) (*Gauss-Jordan-in-ij A* (*from-nat 0*) (*from-nat k*))
using *condition-1-part-2*[*OF j-not-0 all-zero Amk-not-zero*] **.**
next
fix *m* **assume** *not-zero-mkA*: ¬ *is-zero-row-upt-k m k A*
and *zero-below-greatest*: $\forall$ *m*$\geq$(*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*. *A $ m $ from-nat k = 0*
show *is-zero-row-upt-k j* (*Suc k*) *A* **using** *condition-1-part-3*[*OF rref i-less-j not-zero-mkA zero-below-greatest*] *zero-i-suc-k*
unfolding *B ia* **.**
next
fix *m* **assume** *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
and *greatest-eq-card*: *Suc* (*to-nat* (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*)) = *nrows A*
show *is-zero-row-upt-k j* (*Suc k*) *A*
using *condition-1-part-4*[*OF rref - i-less-j not-zero-m greatest-eq-card*] *zero-i-suc-k*
unfolding *B ia nrows-def* **.**
next
fix *m ma*
assume *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
and *greatest-not-card*: *Suc* (*to-nat* (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*)) $\neq$ *nrows A*
and *greatest-less-ma*: (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1* $\leq$ *ma*
and *A-ma-k-not-zero*: *A $ ma $ from-nat k* $\neq$ *0*
show *is-zero-row-upt-k j* (*Suc k*) (*Gauss-Jordan-in-ij A* ((*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*))
using *condition-1-part-5*[*OF rref - i-less-j not-zero-m greatest-not-card greatest-less-ma A-ma-k-not-zero*]
using *zero-i-suc-k*
unfolding *B ia* **.**
qed

lemma *condition-2-part-1*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*if* $\forall$ *m. is-zero-row-upt-k m k A then 0 else to-nat* (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*)
defines *B*:*B*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
and *all-zero*: $\forall$ *m. is-zero-row-upt-k m k A*
and *all-zero-k*: $\forall$ *m. A $ m $ from-nat k = 0*
shows *A $ i $* (*LEAST k. A $ i $ k* $\neq$ *0*) = *1*
proof −
have *ia2*: *ia = 0* **using** *ia all-zero* **by** *simp*
have *B-eq-A*: *B*=*A* **unfolding** *B Gauss-Jordan-column-k-def Let-def fst-conv*

snd-conv ia2 **using** *all-zero-k* **by** *fastforce*
show *?thesis* **using** *all-zero-k condition-1-part-1*[*OF - all-zero*] *not-zero-i-suc-k*
unfolding *B-eq-A* **by** *presburger*
qed

lemma *condition-2-part-2*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*if* ∀ *m*. *is-zero-row-upt-k m k A then 0 else to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1*)
defines *B*:*B*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
and *all-zero*: ∀ *m*. *is-zero-row-upt-k m k A*
and *Amk-not-zero*: *A* $ *m* $ *from-nat k* ≠ *0*
shows *Gauss-Jordan-in-ij A 0* (*from-nat k*) $ *i* $ (*LEAST ka*. *Gauss-Jordan-in-ij A 0* (*from-nat k*) $ *i* $ *ka* ≠ *0*) = *1*
proof −
have *ia2*: *ia* = *0* **unfolding** *ia* **using** *all-zero* **by** *simp*
have *B-eq*: *B* = *Gauss-Jordan-in-ij A 0* (*from-nat k*) **unfolding** *B Gauss-Jordan-column-k-def*
unfolding *ia2 Let-def fst-conv snd-conv*
using *Amk-not-zero least-mod-type* **unfolding** *from-nat-0 nrows-def* **by** *auto*
have *i-eq-0*: *i*=*0* **using** *Amk-not-zero B-eq all-zero condition-1-part-2 from-nat-0 not-zero-i-suc-k* **by** *metis*
have *Least-eq*: (*LEAST ka*. *Gauss-Jordan-in-ij A 0* (*from-nat k*) $ *i* $ *ka* ≠ *0*) = *from-nat k*
proof (*rule Least-equality*)
have *Gauss-Jordan-in-ij A 0* (*from-nat k*) $ *0* $ *from-nat k* = *1* **using** *Gauss-Jordan-in-ij-1 Amk-not-zero least-mod-type* **by** *blast*
thus *Gauss-Jordan-in-ij A 0* (*from-nat k*) $ *i* $ *from-nat k* ≠ *0* **unfolding** *i-eq-0* **by** *simp*
fix *y* **assume** *not-zero-gauss*: *Gauss-Jordan-in-ij A 0* (*from-nat k*) $ *i* $ *y* ≠ *0*
show *from-nat k* ≤ *y*
proof (*rule ccontr*)
assume ¬ *from-nat k* ≤ *y* **hence** *y*: *y* < *from-nat k* **by** *force*
have *Gauss-Jordan-in-ij A 0* (*from-nat k*) $ *0* $ *y* = *A* $ *0* $ *y*
by (*rule Gauss-Jordan-in-ij-preserves-previous-elements'*[*OF all-zero to-nat-le*[*OF y*] *Amk-not-zero*])
also have ... = *0* **using** *all-zero to-nat-le*[*OF y*] **unfolding** *is-zero-row-upt-k-def* **by** *blast*
finally show *False* **using** *not-zero-gauss* **unfolding** *i-eq-0* **by** *contradiction*
qed
qed
show *?thesis* **unfolding** *Least-eq* **unfolding** *i-eq-0* **by** (*rule Gauss-Jordan-in-ij-1*, *auto intro*!: *Amk-not-zero least-mod-type*)
qed

lemma *condition-2-part-3*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*if* ∀ *m*. *is-zero-row-upt-k m k A then 0 else to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1*)
defines *B*:*B*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *rref*: *reduced-row-echelon-form-upt-k A k*
and *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
and *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
and *zero-below-greatest*: ∀ *m*≥(*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1*. *A* $ *m* $ *from-nat k* = *0*
shows *A* $ *i* $ (*LEAST k*. *A* $ *i* $ *k* ≠ *0*) = *1*
proof −
have *ia2*: *ia*=*to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1* **unfolding** *ia* **using** *not-zero-m* **by** *presburger*
have *B-eq-A*: *B*=*A*
unfolding *B Gauss-Jordan-column-k-def Let-def fst-conv snd-conv ia2*
apply *simp*
unfolding *from-nat-to-nat-greatest* **using** *zero-below-greatest* **by** *blast*
show *?thesis*
proof (*cases to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1* < *CARD*(*'rows*))
case *True*
have ¬ *is-zero-row-upt-k i k A*
proof −
have *i*<(*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1*
proof (*rule ccontr*)
assume ¬ *i* < (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1*
hence *i*: (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1* ≤ *i* **by** *simp*
hence (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) < *i* **using** *le-Suc' True* **by** *simp*
hence *zero-i*: *is-zero-row-upt-k i k A* **using** *not-greater-Greatest* **by** *blast*
hence *is-zero-row-upt-k i* (*Suc k*) *A*
proof (*unfold is-zero-row-upt-k-def*, *clarify*)
fix *j*::*'columns*
assume *to-nat j* < *Suc k*
thus *A* $ *i* $ *j* = *0*
using *zero-i* **unfolding** *is-zero-row-upt-k-def* **using** *zero-below-greatest i*
by (*metis from-nat-to-nat-id le-neq-implies-less not-le not-less-eq-eq*)
qed
thus *False* **using** *not-zero-i-suc-k* **unfolding** *B-eq-A* **by** *contradiction*
qed
hence *i*≤(*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) **using** *not-le le-Suc* **by** *metis*
thus *?thesis* **using** *greatest-ge-nonzero-row'*[*OF rref*] *not-zero-m* **by** *fast*
qed
thus *?thesis* **using** *rref-upt-condition2*[*OF rref*] **by** *blast*
next
case *False*
have *greatest-plus-one-eq-0*: (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1* = *0*

```
    using to-nat-plus-one-less-card False by blast
   have ¬ is-zero-row-upt-k i k A
   proof (rule not-is-zero-row-upt-suc)
    show ¬ is-zero-row-upt-k i (Suc k) A using not-zero-i-suc-k unfolding B-eq-A
.
     show ∀ i. A $ i $ from-nat k = 0
      using zero-below-greatest
      unfolding greatest-plus-one-eq-0 using least-mod-type by blast
   qed
   thus ?thesis using rref-upt-condition2[OF rref] by blast
  qed
qed

lemma condition-2-part-4:
  fixes A::'a::{field}^'columns::{mod-type}^'rows::{mod-type} and k::nat
  assumes rref: reduced-row-echelon-form-upt-k A k
  and not-zero-m: ¬ is-zero-row-upt-k m k A
  and greatest-eq-card: Suc (to-nat (GREATEST n. ¬ is-zero-row-upt-k n k A)) =
nrows A
  shows A $ i $ (LEAST k. A $ i $ k ≠ 0) = 1
proof −
  have ¬ is-zero-row-upt-k i k A
  proof (rule ccontr, simp)
   assume zero-i: is-zero-row-upt-k i k A
   hence zero-minus-1: is-zero-row-upt-k (−1) k A
    using rref-upt-condition1[OF rref]
    using Greatest-is-minus-1 neq-le-trans by metis
    have (GREATEST n. ¬ is-zero-row-upt-k n k A) + 1 = 0 using great-
est-plus-one-eq-0[OF greatest-eq-card] .
   hence greatest-eq-minus-1: (GREATEST n. ¬ is-zero-row-upt-k n k A) = −1
using a-eq-minus-1 by fast
   have ¬ is-zero-row-upt-k (GREATEST n. ¬ is-zero-row-upt-k n k A) k A
    by (rule greatest-ge-nonzero-row'[OF rref - ], auto intro!: not-zero-m)
  thus False using zero-minus-1 unfolding greatest-eq-minus-1 by contradiction
  qed
  thus ?thesis using rref-upt-condition2[OF rref] by blast
qed


lemma condition-2-part-5:
  fixes A::'a::{field}^'columns::{mod-type}^'rows::{mod-type} and k::nat
  defines ia:ia≡(if ∀ m. is-zero-row-upt-k m k A then 0 else to-nat (GREATEST
n. ¬ is-zero-row-upt-k n k A) + 1)
  defines B:B≡(snd (Gauss-Jordan-column-k (ia,A) k))
  assumes rref: reduced-row-echelon-form-upt-k A k
  and not-zero-i-suc-k: ¬ is-zero-row-upt-k i (Suc k) B
  and not-zero-m: ¬ is-zero-row-upt-k m k A
  and greatest-noteq-card: Suc (to-nat (GREATEST n. ¬ is-zero-row-upt-k n k A))
≠ nrows A
```

and *greatest-less-ma*: (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1* ≤ *ma*
and *A-ma-k-not-zero*: *A* $ *ma* $ *from-nat k* ≠ *0*
shows *Gauss-Jordan-in-ij A* ((*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *i* $
(*LEAST ka. Gauss-Jordan-in-ij A* ((*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *i* $ *ka* ≠ *0*) = *1*
proof −
have *ia2*: *ia=to-nat* (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1* **unfolding** *ia* **using** *not-zero-m* **by** *presburger*
have *B-eq-Gauss*: *B=Gauss-Jordan-in-ij A* ((*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*)
unfolding *B Gauss-Jordan-column-k-def Let-def fst-conv snd-conv ia2*
apply *simp*
unfolding *from-nat-to-nat-greatest* **using** *greatest-noteq-card A-ma-k-not-zero greatest-less-ma* **by** *blast*
have *greatest-plus-one-not-zero*: (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1* ≠ *0*
using *suc-not-zero greatest-noteq-card* **unfolding** *nrows-def* **by** *auto*
show *?thesis*
proof (*cases is-zero-row-upt-k i k A*)
case *True*
hence *not-zero-iB*: *is-zero-row-upt-k i k B* **unfolding** *is-zero-row-upt-k-def* **unfolding** *B-eq-Gauss*
using *Gauss-Jordan-in-ij-preserves-previous-elements*[*OF rref not-zero-m - greatest-plus-one-not-zero*]
using *A-ma-k-not-zero greatest-less-ma* **by** *fastforce*
hence *Gauss-Jordan-i-not-0*: *Gauss-Jordan-in-ij A* ((*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *i* $ (*from-nat k*) ≠ *0*
using *not-zero-i-suc-k* **unfolding** *B-eq-Gauss* **unfolding** *is-zero-row-upt-k-def* **using** *from-nat-to-nat-id less-Suc-eq* **by** (*metis* (*lifting*, *no-types*))
have *i* = ((*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*)
proof (*rule ccontr*)
assume *i-not-greatest*: *i* ≠ (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*
have *Gauss-Jordan-in-ij A* ((*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *i* $ (*from-nat k*) = *0*
proof (*rule Gauss-Jordan-in-ij-0*)
show ∃ *n. A* $ *n* $ *from-nat k* ≠ *0* ∧ (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1* ≤ *n* **using** *A-ma-k-not-zero greatest-less-ma* **by** *blast*
show *i* ≠ (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1* **using** *i-not-greatest*
.
qed
thus *False* **using** *Gauss-Jordan-i-not-0* **by** *contradiction*
qed
hence *Gauss-Jordan-i-1*: *Gauss-Jordan-in-ij A* ((*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *i* $ (*from-nat k*) = *1*
using *Gauss-Jordan-in-ij-1* **using** *A-ma-k-not-zero greatest-less-ma* **by** *blast*
have *Least-eq-k*: (*LEAST ka. Gauss-Jordan-in-ij A* ((*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *i* $ *ka* ≠ *0*) = *from-nat k*
proof (*rule Least-equality*)

show *Gauss-Jordan-in-ij A* ((*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *i* $ *from-nat k* ≠ *0* **using** *Gauss-Jordan-i-not-0* **.**
show ⋀*y. Gauss-Jordan-in-ij A* ((*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *i* $ *y* ≠ *0* ⟹ *from-nat k* ≤ *y*
using *B-eq-Gauss is-zero-row-upt-k-def not-less not-zero-iB to-nat-le* **by** *fast*
qed
show *?thesis* **using** *Gauss-Jordan-i-1* **unfolding** *Least-eq-k* **.**
next
case *False*
obtain *j* **where** *Aij-not-0*: *A* $ *i* $ *j* ≠ *0* **and** *j-le-k*: *to-nat j* < *k* **using** *False* **unfolding** *is-zero-row-upt-k-def* **by** *auto*
have *least-le-k*: *to-nat* (*LEAST ka. A* $ *i* $ *ka* ≠ *0*) < *k*
by (*metis* (*lifting*, *mono-tags*) *Aij-not-0 j-le-k less-trans linorder-cases not-less-Least to-nat-mono*)
have *least-le-j*: (*LEAST ka. Gauss-Jordan-in-ij A* ((*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *i* $ *ka* ≠ *0*) ≤ *j*
using *Gauss-Jordan-in-ij-preserves-previous-elements*[*OF rref not-zero-m - greatest-plus-one-not-zero j-le-k*] **using** *A-ma-k-not-zero greatest-less-ma*
using *Aij-not-0 False not-le-imp-less not-less-Least* **by** (*metis* (*mono-tags*))
have *Least-eq*: (*LEAST ka. Gauss-Jordan-in-ij A* ((*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *i* $ *ka* ≠ *0*)
= (*LEAST ka. A* $ *i* $ *ka* ≠ *0*)
proof (*rule Least-equality*)
show *Gauss-Jordan-in-ij A* ((*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *i* $ (*LEAST ka. A* $ *i* $ *ka* ≠ *0*) ≠ *0*
using *Gauss-Jordan-in-ij-preserves-previous-elements*[*OF rref False - greatest-plus-one-not-zero*] *least-le-k False rref-upt-condition2*[*OF rref*]
using *A-ma-k-not-zero B-eq-Gauss greatest-less-ma zero-neq-one* **by** *fastforce*
fix *y* **assume** *Gauss-Jordan-y*:*Gauss-Jordan-in-ij A* ((*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *i* $ *y* ≠ *0*
show (*LEAST ka. A* $ *i* $ *ka* ≠ *0*) ≤ *y*
proof (*cases to-nat y* < *k*)
case *False*
thus *?thesis*
using *least-le-k less-trans not-le-imp-less to-nat-from-nat to-nat-le* **by** *metis*
next
case *True*
have *A* $ *i* $ *y* ≠ *0* **using** *Gauss-Jordan-y* **using** *Gauss-Jordan-in-ij-preserves-previous-elements*[*OF rref not-zero-m - greatest-plus-one-not-zero True*]
using *A-ma-k-not-zero greatest-less-ma* **by** *fastforce*
thus *?thesis* **using** *Least-le* **by** *fastforce*
qed
qed
have *A* $ *i* $ (*LEAST ka. Gauss-Jordan-in-ij A* ((*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *i* $ *ka* ≠ *0*) = *1*
using *False* **using** *rref-upt-condition2*[*OF rref*] **unfolding** *Least-eq* **by** *blast*

thus *?thesis* **unfolding** *Least-eq* **using** *Gauss-Jordan-in-ij-preserves-previous-elements*[*OF rref False - greatest-plus-one-not-zero*]

using *least-le-k A-ma-k-not-zero greatest-less-ma* **by** *fastforce*
qed
qed

lemma *condition-2*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*if* ∀ *m*. *is-zero-row-upt-k m k A then 0 else to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1*)
defines *B*:*B*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *rref*: *reduced-row-echelon-form-upt-k A k*
and *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
shows *B* $ *i* $ (*LEAST k*. *B* $ *i* $ *k* ≠ *0*) = *1*
proof (*unfold B Gauss-Jordan-column-k-def ia Let-def fst-conv snd-conv*, *auto*, *unfold from-nat-to-nat-greatest from-nat-0*)
assume *all-zero*: ∀ *m*. *is-zero-row-upt-k m k A* **and** *all-zero-k*: ∀ *m*≥*0*. *A* $ *m* $ *from-nat k* = *0*
show *A* $ *i* $ (*LEAST k*. *A* $ *i* $ *k* ≠ *0*) = *1*
using *condition-2-part-1*[*OF - all-zero*] *not-zero-i-suc-k all-zero-k least-mod-type* **unfolding** *B ia* **by** *blast*
next
fix *m* **assume** *all-zero*: ∀ *m*. *is-zero-row-upt-k m k A*
and *Amk-not-zero*: *A* $ *m* $ *from-nat k* ≠ *0*
show *Gauss-Jordan-in-ij A 0* (*from-nat k*) $ *i* $ (*LEAST ka*. *Gauss-Jordan-in-ij A 0* (*from-nat k*) $ *i* $ *ka* ≠ *0*) = *1*
using *condition-2-part-2*[*OF - all-zero Amk-not-zero*] *not-zero-i-suc-k* **unfolding** *B ia* **.**
next
fix *m*
assume *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
and *zero-below-greatest*: ∀ *m*≥(*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1*. *A* $ *m* $ *from-nat k* = *0*
show *A* $ *i* $ (*LEAST k*. *A* $ *i* $ *k* ≠ *0*) = *1* **using** *condition-2-part-3*[*OF rref - not-zero-m zero-below-greatest*] *not-zero-i-suc-k* **unfolding** *B ia* **.**
next
fix *m*
assume *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
and *greatest-eq-card*: *Suc* (*to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*)) = *nrows A*
show *A* $ *i* $ (*LEAST k*. *A* $ *i* $ *k* ≠ *0*) = *1* **using** *condition-2-part-4*[*OF rref not-zero-m greatest-eq-card*] **.**
next
fix *m ma*
assume *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
and *greatest-noteq-card*: *Suc* (*to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*)) ≠ *nrows A*
and *greatest-less-ma*: (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1* ≤ *ma*
and *A-ma-k-not-zero*: *A* $ *ma* $ *from-nat k* ≠ *0*
show *Gauss-Jordan-in-ij A* ((*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1*)

(*from-nat k*) $ *i*
$ (*LEAST ka. Gauss-Jordan-in-ij A* ((*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *i* $ *ka* ≠ *0*) = *1*
using *condition-2-part-5*[*OF rref - not-zero-m greatest-noteq-card greatest-less-ma A-ma-k-not-zero*] *not-zero-i-suc-k* **unfolding** *B ia* **.**
qed

lemma *condition-3-part-1*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*if* ∀ *m. is-zero-row-upt-k m k A then 0 else to-nat* (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*)
defines *B*:*B*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
and *all-zero*: ∀ *m. is-zero-row-upt-k m k A*
and *all-zero-k*: ∀ *m. A* $ *m* $ *from-nat k* = *0*
shows (*LEAST n. A* $ *i* $ *n* ≠ *0*) < (*LEAST n. A* $ (*i* + *1*) $ *n* ≠ *0*)
proof −
have *ia2*: *ia* = *0* **using** *ia all-zero* **by** *simp*
have *B-eq-A*: *B*=*A* **unfolding** *B Gauss-Jordan-column-k-def Let-def fst-conv snd-conv ia2* **using** *all-zero-k* **by** *fastforce*
have *is-zero-row-upt-k i* (*Suc k*) *B* **using** *all-zero all-zero-k* **unfolding** *B-eq-A is-zero-row-upt-k-def* **by** (*metis less-SucE to-nat-from-nat*)
thus *?thesis* **using** *not-zero-i-suc-k* **by** *contradiction*
qed

lemma *condition-3-part-2*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*if* ∀ *m. is-zero-row-upt-k m k A then 0 else to-nat* (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*)
defines *B*:*B*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *i-le*: *i* < *i* + *1*
and *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
and *not-zero-suc-i-suc-k*: ¬ *is-zero-row-upt-k* (*i* + *1*) (*Suc k*) *B*
and *all-zero*: ∀ *m. is-zero-row-upt-k m k A*
and *Amk-notzero*: *A* $ *m* $ *from-nat k* ≠ *0*
shows (*LEAST n. Gauss-Jordan-in-ij A 0* (*from-nat k*) $ *i* $ *n* ≠ *0*) < (*LEAST n. Gauss-Jordan-in-ij A 0* (*from-nat k*) $ (*i* + *1*) $ *n* ≠ *0*)
proof −
have *ia2*: *ia* = *0* **using** *ia all-zero* **by** *simp*
have *B-eq-Gauss*: *B* = *Gauss-Jordan-in-ij A 0* (*from-nat k*)
unfolding *B Gauss-Jordan-column-k-def Let-def fst-conv snd-conv ia2*
using *all-zero Amk-notzero least-mod-type* **unfolding** *from-nat-0* **by** *auto*
have *i=0* **using** *all-zero-imp-Gauss-Jordan-column-not-zero-in-row-0*[*OF all-zero - Amk-notzero*] *not-zero-i-suc-k* **unfolding** *B ia* **.**
moreover have *i+1=0* **using** *all-zero-imp-Gauss-Jordan-column-not-zero-in-row-0*[*OF all-zero - Amk-notzero*] *not-zero-suc-i-suc-k* **unfolding** *B ia* **.**

ultimately show *?thesis* **using** *i-le* **by** *auto*
qed

lemma *condition-3-part-3*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*if* ∀ *m*. *is-zero-row-upt-k m k A then 0 else to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1*)
defines *B*:*B*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *rref*: *reduced-row-echelon-form-upt-k A k*
and *i-le*: *i* < *i* + *1*
and *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
and *not-zero-suc-i-suc-k*: ¬ *is-zero-row-upt-k* (*i* + *1*) (*Suc k*) *B*
and *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
and *zero-below-greatest*: ∀ *m*≥(*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1*. *A* $ *m* $ *from-nat k* = *0*
shows (*LEAST n*. *A* $ *i* $ *n* ≠ *0*) < (*LEAST n*. *A* $ (*i* + *1*) $ *n* ≠ *0*)
proof −
have *ia2*: *ia*=*to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1* **unfolding** *ia* **using** *not-zero-m* **by** *presburger*
have *B-eq-A*: *B*=*A*
unfolding *B Gauss-Jordan-column-k-def Let-def fst-conv snd-conv ia2*
apply *simp*
unfolding *from-nat-to-nat-greatest* **using** *zero-below-greatest* **by** *blast*
have *rref-suc*: *reduced-row-echelon-form-upt-k A* (*Suc k*)
proof (*rule rref-suc-if-zero-below-greatest*)
show *reduced-row-echelon-form-upt-k A k* **using** *rref* **.**
show ¬ (∀ *a*. *is-zero-row-upt-k a k A*) **using** *not-zero-m* **by** *fast*
show ∀ *a*>*GREATEST m*. ¬ *is-zero-row-upt-k m k A*. *is-zero-row-upt-k a* (*Suc k*) *A*
proof (*clarify*)
fix *a*::*'rows* **assume** *greatest-less-a*: (*GREATEST m*. ¬ *is-zero-row-upt-k m k A*) < *a*
show *is-zero-row-upt-k a* (*Suc k*) *A*
proof (*rule is-zero-row-upt-k-suc*)
show *is-zero-row-upt-k a k A* **using** *greatest-less-a row-greater-greatest-is-zero* **by** *fast*
show *A* $ *a* $ *from-nat k* = *0* **using** *le-Suc*[*OF greatest-less-a*] *zero-below-greatest* **by** *fast*
qed
qed
qed
show *?thesis* **using** *rref-upt-condition3*[*OF rref-suc*] *i-le not-zero-i-suc-k not-zero-suc-i-suc-k* **unfolding** *B-eq-A* **by** *blast*
qed

lemma *condition-3-part-4*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*if* ∀ *m*. *is-zero-row-upt-k m k A then 0 else to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1*)
defines *B*:*B*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *rref*: *reduced-row-echelon-form-upt-k A k* **and** *i-le*: *i* < *i* + *1*
and *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
and *not-zero-suc-i-suc-k*: ¬ *is-zero-row-upt-k* (*i* + *1*) (*Suc k*) *B*
and *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
and *greatest-eq-card*: *Suc* (*to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*)) = *nrows A*
shows (*LEAST n*. *A* $ *i* $ *n* ≠ *0*) < (*LEAST n*. *A* $ (*i* + *1*) $ *n* ≠ *0*)
proof −
have *ia2*: *ia*=*to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1* **unfolding** *ia* **using** *not-zero-m* **by** *presburger*
have *B-eq-A*: *B*=*A*
unfolding *B Gauss-Jordan-column-k-def Let-def fst-conv snd-conv ia2*
unfolding *from-nat-to-nat-greatest* **using** *greatest-eq-card* **by** *simp*
have *greatest-eq-minus-1*: (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) = −*1*
using *a-eq-minus-1 greatest-eq-card to-nat-plus-one-less-card* **unfolding** *nrows-def* **by** *fastforce*
have *rref-suc*: *reduced-row-echelon-form-upt-k A* (*Suc k*)
proof (*rule rref-suc-if-all-rows-not-zero*)
show *reduced-row-echelon-form-upt-k A k* **using** *rref* **.**
show ∀ *n*. ¬ *is-zero-row-upt-k n k A* **using** *Greatest-is-minus-1 greatest-eq-minus-1 greatest-ge-nonzero-row'*[*OF rref* -] *not-zero-m* **by** *metis*
qed
show *?thesis* **using** *rref-upt-condition3*[*OF rref-suc*] *i-le not-zero-i-suc-k not-zero-suc-i-suc-k* **unfolding** *B-eq-A* **by** *blast*
qed

lemma *condition-3-part-5*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*if* ∀ *m*. *is-zero-row-upt-k m k A then 0 else to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1*)
defines *B*:*B*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *rref*: *reduced-row-echelon-form-upt-k A k*
and *i-le*: *i* < *i* + *1*
and *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
and *not-zero-suc-i-suc-k*: ¬ *is-zero-row-upt-k* (*i* + *1*) (*Suc k*) *B*
and *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
and *greatest-not-card*: *Suc* (*to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*)) ≠ *nrows A*
and *greatest-less-ma*: (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1* ≤ *ma*
and *A-ma-k-not-zero*: *A* $ *ma* $ *from-nat k* ≠ *0*
shows (*LEAST n*. *Gauss-Jordan-in-ij A* ((*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *i* $ *n* ≠ *0*)
< (*LEAST n*. *Gauss-Jordan-in-ij A* ((*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ (*i* + *1*) $ *n* ≠ *0*)

proof −
have *ia2*: *ia=to-nat* (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1* **unfolding** *ia* **using** *not-zero-m* **by** *presburger*
have *B-eq-Gauss*: *B* = *Gauss-Jordan-in-ij A* ((*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*)
unfolding *B Gauss-Jordan-column-k-def*
unfolding *ia2 Let-def fst-conv snd-conv*
using *greatest-not-card greatest-less-ma A-ma-k-not-zero*
by (*auto simp add*: *from-nat-to-nat-greatest*)
have *suc-greatest-not-zero*: (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1* ≠ *0*
using *Suc-eq-plus1 suc-not-zero greatest-not-card* **unfolding** *nrows-def* **by** *auto*
show *?thesis*
proof (*cases is-zero-row-upt-k* (*i* + *1*) *k A*)
case *True*
have *zero-i-plus-one-k-B*: *is-zero-row-upt-k* (*i*+*1*) *k B*
by (*unfold B-eq-Gauss*, *rule is-zero-after-Gauss*[*OF True not-zero-m rref greatest-less-ma A-ma-k-not-zero*])
hence *Gauss-Jordan-i-not-0*: *Gauss-Jordan-in-ij A* ((*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ (*i*+*1*) $ (*from-nat k*) ≠ *0*
using *not-zero-suc-i-suc-k* **unfolding** *B-eq-Gauss* **using** *is-zero-row-upt-k-suc* **by** *blast*
have *i-plus-one-eq*: *i* + *1* = ((*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*)
proof (*rule ccontr*)
assume *i-not-greatest*: *i* + *1* ≠ (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*
have *Gauss-Jordan-in-ij A* ((*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ (*i* + *1*) $ (*from-nat k*) = *0*
proof (*rule Gauss-Jordan-in-ij-0*)
show ∃ *n. A* $ *n* $ *from-nat k* ≠ *0* ∧ (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1* ≤ *n* **using** *greatest-less-ma A-ma-k-not-zero* **by** *blast*
show *i* + *1* ≠ (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1* **using** *i-not-greatest* **.**
qed
thus *False* **using** *Gauss-Jordan-i-not-0* **by** *contradiction*
qed
hence *i-eq-greatest*: *i*=(*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) **using** *add-right-cancel* **by** *simp*
have *Least-eq-k*: (*LEAST ka. Gauss-Jordan-in-ij A* ((*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ (*i*+*1*) $ *ka* ≠ *0*) = *from-nat k*
proof (*rule Least-equality*)
show *Gauss-Jordan-in-ij A* ((*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ (*i*+*1*) $ *from-nat k* ≠ *0* **by** (*metis Gauss-Jordan-i-not-0*)
fix *y* **assume** *Gauss-Jordan-in-ij A* ((*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ (*i*+*1*) $ *y* ≠ *0*
thus *from-nat k* ≤ *y* **using** *zero-i-plus-one-k-B* **unfolding** *i-eq-greatest B-eq-Gauss* **by** (*metis is-zero-row-upt-k-def not-less to-nat-le*)
qed
have *not-zero-i-A*: ¬ *is-zero-row-upt-k i k A* **using** *greatest-less-zero-row*[*OF rref*] *not-zero-m* **unfolding** *i-eq-greatest* **by** *fast*

from *this* **obtain** *j* **where** *Aij-not-0*: *A $ i $ j ≠ 0* **and** *j-le-k*: *to-nat j < k*
unfolding *is-zero-row-upt-k-def* **by** *blast*
have *least-le-k*: *to-nat* (*LEAST ka. A $ i $ ka ≠ 0*) < *k*
by (*metis* (*lifting*, *mono-tags*) *Aij-not-0 j-le-k less-trans linorder-cases not-less-Least to-nat-mono*)
have *Least-eq*: (*LEAST n. Gauss-Jordan-in-ij A* ((*GREATEST n. ¬ is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *i* $ *n ≠ 0*) =
(*LEAST n. A $ i $ n ≠ 0*)
proof (*rule Least-equality*)
show *Gauss-Jordan-in-ij A* ((*GREATEST n. ¬ is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *i* $ (*LEAST ka. A $ i $ ka ≠ 0*) ≠ *0*
using *Gauss-Jordan-in-ij-preserves-previous-elements*[*OF rref not-zero-i-A - suc-greatest-not-zero least-le-k*] *greatest-less-ma A-ma-k-not-zero*
using *rref-upt-condition2*[*OF rref*] *not-zero-i-A* **by** *fastforce*
fix *y* **assume** *Gauss-Jordan-y*:*Gauss-Jordan-in-ij A* ((*GREATEST n. ¬ is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *i* $ *y ≠ 0*
show (*LEAST ka. A $ i $ ka ≠ 0*) ≤ *y*
proof (*cases to-nat y < k*)
case *False* **thus** *?thesis* **by** (*metis not-le least-le-k less-trans to-nat-mono*)
next
case *True*
have *A $ i $ y ≠ 0* **using** *Gauss-Jordan-y* **using** *Gauss-Jordan-in-ij-preserves-previous-elements*[*OF rref not-zero-m - suc-greatest-not-zero True*]
using *A-ma-k-not-zero greatest-less-ma* **by** *fastforce*
thus *?thesis* **using** *Least-le* **by** *fastforce*
qed
qed
also have *... < from-nat k* **by** (*metis is-zero-row-upt-k-def is-zero-row-upt-k-suc le-less-linear le-less-trans least-le-k not-zero-suc-i-suc-k to-nat-mono′ zero-i-plus-one-k-B*)

finally show *?thesis* **unfolding** *Least-eq-k* **.**
next
case *False*
have *not-zero-i-A*: *¬ is-zero-row-upt-k i k A* **using** *rref-upt-condition1*[*OF rref*] *False i-le* **by** *blast*
from *this* **obtain** *j* **where** *Aij-not-0*: *A $ i $ j ≠ 0* **and** *j-le-k*: *to-nat j < k*
unfolding *is-zero-row-upt-k-def* **by** *blast*
have *least-le-k*: *to-nat* (*LEAST ka. A $ i $ ka ≠ 0*) < *k*
by (*metis* (*lifting*, *mono-tags*) *Aij-not-0 j-le-k less-trans linorder-cases not-less-Least to-nat-mono*)
have *Least-i-eq*: (*LEAST n. Gauss-Jordan-in-ij A* ((*GREATEST n. ¬ is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *i* $ *n ≠ 0*)
= (*LEAST n. A $ i $ n ≠ 0*)
proof (*rule Least-equality*)
show *Gauss-Jordan-in-ij A* ((*GREATEST n. ¬ is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *i* $ (*LEAST ka. A $ i $ ka ≠ 0*) ≠ *0*
using *Gauss-Jordan-in-ij-preserves-previous-elements*[*OF rref not-zero-i-A - suc-greatest-not-zero least-le-k*] *greatest-less-ma A-ma-k-not-zero*
using *rref-upt-condition2*[*OF rref*] *not-zero-i-A* **by** *fastforce*

fix *y* **assume** *Gauss-Jordan-y*:*Gauss-Jordan-in-ij A* ((*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *i* $ *y* ≠ *0*
show (*LEAST ka*. *A* $ *i* $ *ka* ≠ *0*) ≤ *y*
proof (*cases to-nat y* < *k*)
case *False* **thus** *?thesis* **by** (*metis not-le not-less-iff-gr-or-eq le-less-trans least-le-k to-nat-mono*)
next
case *True*
have *A* $ *i* $ *y* ≠ *0* **using** *Gauss-Jordan-y* **using** *Gauss-Jordan-in-ij-preserves-previous-elements*[*OF rref not-zero-m - suc-greatest-not-zero True*]
using *A-ma-k-not-zero greatest-less-ma* **by** *fastforce*
thus *?thesis* **using** *Least-le* **by** *fastforce*
qed
qed
from *False* **obtain** *s* **where** *Ais-not-0*: *A* $ (*i*+*1*) $ *s* ≠ *0* **and** *s-le-k*: *to-nat s* < *k* **unfolding** *is-zero-row-upt-k-def* **by** *blast*
have *least-le-k*: *to-nat* (*LEAST ka*. *A* $ (*i*+*1*) $ *ka* ≠ *0*) < *k*
by (*metis* (*lifting*, *mono-tags*) *Ais-not-0 s-le-k neq-iff less-trans not-less-Least to-nat-mono*)
have *Least-i-plus-one-eq*: (*LEAST n*. *Gauss-Jordan-in-ij A* ((*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ (*i*+*1*) $ *n* ≠ *0*)
= (*LEAST n*. *A* $ (*i*+*1*) $ *n* ≠ *0*)
proof (*rule Least-equality*)
show *Gauss-Jordan-in-ij A* ((*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ (*i*+*1*) $ (*LEAST ka*. *A* $ (*i*+*1*) $ *ka* ≠ *0*) ≠ *0*
using *Gauss-Jordan-in-ij-preserves-previous-elements*[*OF rref not-zero-i-A - suc-greatest-not-zero least-le-k*] *greatest-less-ma A-ma-k-not-zero*
using *rref-upt-condition2*[*OF rref*] *False* **by** *fastforce*
fix *y* **assume** *Gauss-Jordan-y*:*Gauss-Jordan-in-ij A* ((*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ (*i*+*1*) $ *y* ≠ *0*
show (*LEAST ka*. *A* $ (*i*+*1*) $ *ka* ≠ *0*) ≤ *y*
proof (*cases to-nat y* < *k*)
case *False* **thus** *?thesis* **by** (*metis* (*mono-tags*) *le-less-linear least-le-k less-trans to-nat-mono*)
next
case *True*
have *A* $ (*i*+*1*) $ *y* ≠ *0* **using** *Gauss-Jordan-y* **using** *Gauss-Jordan-in-ij-preserves-previous-elements*[*OF rref not-zero-m - suc-greatest-not-zero True*]
using *A-ma-k-not-zero greatest-less-ma* **by** *fastforce*
thus *?thesis* **using** *Least-le* **by** *fastforce*
qed
qed
show *?thesis* **unfolding** *Least-i-plus-one-eq Least-i-eq* **using** *rref-upt-condition3*[*OF rref*] *i-le False not-zero-i-A* **by** *blast*
qed
qed

lemma *condition-3*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*

defines *ia*:*ia*≡(*if* ∀ *m*. *is-zero-row-upt-k m k A then 0 else to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1*)
defines *B*:*B*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *rref*: *reduced-row-echelon-form-upt-k A k*
and *i-le*: *i* < *i* + *1*
and *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
and *not-zero-suc-i-suc-k*: ¬ *is-zero-row-upt-k* (*i* + *1*) (*Suc k*) *B*
shows (*LEAST n*. *B* $ *i* $ *n* ≠ *0*) < (*LEAST n*. *B* $ (*i* + *1*) $ *n* ≠ *0*)
proof (*unfold B Gauss-Jordan-column-k-def ia Let-def fst-conv snd-conv*, *auto*, *unfold from-nat-to-nat-greatest from-nat-0*)
assume *all-zero*: ∀ *m*. *is-zero-row-upt-k m k A*
and *all-zero-k*: ∀ *m*≥*0*. *A* $ *m* $ *from-nat k* = *0*
show (*LEAST n*. *A* $ *i* $ *n* ≠ *0*) < (*LEAST n*. *A* $ (*i* + *1*) $ *n* ≠ *0*)
using *condition-3-part-1*[*OF - all-zero*] **using** *all-zero-k least-mod-type not-zero-i-suc-k* **unfolding** *B ia* **by** *fast*
next
fix *m* **assume** *all-zero*: ∀ *m*. *is-zero-row-upt-k m k A*
and *Amk-notzero*: *A* $ *m* $ *from-nat k* ≠ *0*
show (*LEAST n*. *Gauss-Jordan-in-ij A 0* (*from-nat k*) $ *i* $ *n* ≠ *0*) < (*LEAST n*. *Gauss-Jordan-in-ij A 0* (*from-nat k*) $ (*i* + *1*) $ *n* ≠ *0*)
using *condition-3-part-2*[*OF i-le - - all-zero Amk-notzero*] **using** *not-zero-i-suc-k not-zero-suc-i-suc-k* **unfolding** *B ia* **.**
next
fix *m*
assume *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
and *zero-below-greatest*: ∀ *m*≥(*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1*. *A* $ *m* $ *from-nat k* = *0*
show (*LEAST n*. *A* $ *i* $ *n* ≠ *0*) < (*LEAST n*. *A* $ (*i* + *1*) $ *n* ≠ *0*)
using *condition-3-part-3*[*OF rref i-le - - not-zero-m zero-below-greatest*] **using** *not-zero-i-suc-k not-zero-suc-i-suc-k* **unfolding** *B ia* **.**
next
fix *m*
assume *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
and *greatest-eq-card*: *Suc* (*to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*)) = *nrows A*
show (*LEAST n*. *A* $ *i* $ *n* ≠ *0*) < (*LEAST n*. *A* $ (*i* + *1*) $ *n* ≠ *0*)
using *condition-3-part-4*[*OF rref i-le - - not-zero-m greatest-eq-card*] **using** *not-zero-i-suc-k not-zero-suc-i-suc-k* **unfolding** *B ia* **.**
next
fix *m ma*
assume *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
and *greatest-not-card*: *Suc* (*to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*)) ≠ *nrows A*
and *greatest-less-ma*: (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1* ≤ *ma*
and *A-ma-k-not-zero*: *A* $ *ma* $ *from-nat k* ≠ *0*
show (*LEAST n*. *Gauss-Jordan-in-ij A* ((*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *i* $ *n* ≠ *0*)
< (*LEAST n*. *Gauss-Jordan-in-ij A* ((*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ (*i* + *1*) $ *n* ≠ *0*)

using *condition-3-part-5*[*OF rref i-le - - not-zero-m greatest-not-card greatest-less-ma A-ma-k-not-zero*]
using *not-zero-i-suc-k not-zero-suc-i-suc-k* **unfolding** *B ia* **.**
qed

lemma *condition-4-part-1*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*if* ∀ *m. is-zero-row-upt-k m k A then 0 else to-nat* (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*)
defines *B*:*B*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
and *all-zero*: ∀ *m. is-zero-row-upt-k m k A*
and *all-zero-k*: ∀ *m. A* $ *m* $ *from-nat k* = *0*
shows *A* $ *j* $ (*LEAST n. A* $ *i* $ *n* ≠ *0*) = *0*
proof −
have *ia2*: *ia* = *0* **using** *ia all-zero* **by** *simp*
have *B-eq-A*: *B*=*A* **unfolding** *B Gauss-Jordan-column-k-def Let-def fst-conv snd-conv ia2* **using** *all-zero-k* **by** *fastforce*
show *?thesis* **using** *B-eq-A all-zero all-zero-k is-zero-row-upt-k-suc not-zero-i-suc-k* **by** *blast*
qed

lemma *condition-4-part-2*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*if* ∀ *m. is-zero-row-upt-k m k A then 0 else to-nat* (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*)
defines *B*:*B*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
and *i-not-j*: *i* ≠ *j*
and *all-zero*: ∀ *m. is-zero-row-upt-k m k A*
and *Amk-not-zero*: *A* $ *m* $ *from-nat k* ≠ *0*
shows *Gauss-Jordan-in-ij A 0* (*from-nat k*) $ *j* $ (*LEAST n. Gauss-Jordan-in-ij A 0* (*from-nat k*) $ *i* $ *n* ≠ *0*) = *0*
proof −
have *i-eq-0*: *i*=*0* **using** *all-zero-imp-Gauss-Jordan-column-not-zero-in-row-0*[*OF all-zero - Amk-not-zero*] *not-zero-i-suc-k* **unfolding** *B ia* **.**
have *least-eq-k*: (*LEAST n. Gauss-Jordan-in-ij A 0* (*from-nat k*) $ *i* $ *n* ≠ *0*) = *from-nat k*
proof (*rule Least-equality*)
show *Gauss-Jordan-in-ij A 0* (*from-nat k*) $ *i* $ *from-nat k* ≠ *0* **unfolding** *i-eq-0* **using** *Amk-not-zero Gauss-Jordan-in-ij-1 least-mod-type zero-neq-one* **by** *fastforce*
fix *y* **assume** *Gauss-Jordan-y-not-0*: *Gauss-Jordan-in-ij A 0* (*from-nat k*) $ *i* $ *y* ≠ *0*
show *from-nat k* ≤ *y*
proof (*rule ccontr*)
assume ¬ *from-nat k* ≤ *y*
hence *y* < (*from-nat k*) **by** *simp*

hence *to-nat-y-less-k*: *to-nat y* < *k* **using** *to-nat-le* **by** *auto*
have *Gauss-Jordan-in-ij A 0* (*from-nat k*) $ *i* $ *y* = *0*
using *Gauss-Jordan-in-ij-preserves-previous-elements'*[*OF all-zero to-nat-y-less-k Amk-not-zero*] *all-zero to-nat-y-less-k*
unfolding *is-zero-row-upt-k-def* **by** *fastforce*
thus *False* **using** *Gauss-Jordan-y-not-0* **by** *contradiction*
qed
qed
show *?thesis* **unfolding** *least-eq-k* **apply** (*rule Gauss-Jordan-in-ij-0*) **using** *i-eq-0 i-not-j Amk-not-zero least-mod-type* **by** *blast+*
qed

lemma *condition-4-part-3*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*if* ∀ *m. is-zero-row-upt-k m k A then 0 else to-nat* (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*)
defines *B*:*B*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *rref*: *reduced-row-echelon-form-upt-k A k*
and *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
and *i-not-j*: *i* ≠ *j*
and *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
and *zero-below-greatest*: ∀ *m*≥(*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1. A* $ *m* $ *from-nat k* = *0*
shows *A* $ *j* $ (*LEAST n. A* $ *i* $ *n* ≠ *0*) = *0*
proof −
have *ia2*: *ia*=*to-nat* (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1* **unfolding** *ia* **using** *not-zero-m* **by** *presburger*
have *B-eq-A*: *B*=*A*
unfolding *B Gauss-Jordan-column-k-def Let-def fst-conv snd-conv ia2*
apply *simp*
unfolding *from-nat-to-nat-greatest* **using** *zero-below-greatest* **by** *blast*
have *rref-suc*: *reduced-row-echelon-form-upt-k A* (*Suc k*)
proof (*rule rref-suc-if-zero-below-greatest*[*OF rref*], *auto intro*!: *not-zero-m*)
fix *a*
assume *greatest-less-a*: (*GREATEST m.* ¬ *is-zero-row-upt-k m k A*) < *a*
show *is-zero-row-upt-k a* (*Suc k*) *A*
proof (*rule is-zero-row-upt-k-suc*)
show *is-zero-row-upt-k a k A* **using** *row-greater-greatest-is-zero*[*OF greatest-less-a*] .
show *A* $ *a* $ *from-nat k* = *0* **using** *zero-below-greatest le-Suc*[*OF greatest-less-a*] **by** *blast*
qed
qed
show *?thesis* **using** *rref-upt-condition4*[*OF rref-suc*] *not-zero-i-suc-k i-not-j* **unfolding** *B-eq-A* **by** *blast*
qed

lemma *condition-4-part-4*:

fixes A::$'a$::{*field*}$^{\prime}$*columns*::{*mod-type*}$^{\prime}$*rows*::{*mod-type*} **and** k::*nat*
defines *ia*:*ia*≡(*if* ∀ *m*. *is-zero-row-upt-k m k A then 0 else to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1*)
defines *B*:*B*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *rref*: *reduced-row-echelon-form-upt-k A k*
and *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
and *i-not-j*: *i* ≠ *j*
and *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
and *greatest-eq-card*: *Suc* (*to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*)) = *nrows A*
shows *A* $ *j* $ (*LEAST n*. *A* $ *i* $ *n* ≠ *0*) = *0*
proof −
have *ia2*: *ia*=*to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1* **unfolding** *ia* **using** *not-zero-m* **by** *presburger*
have *B-eq-A*: *B*=*A*
unfolding *B Gauss-Jordan-column-k-def Let-def fst-conv snd-conv ia2*
unfolding *from-nat-to-nat-greatest* **using** *greatest-eq-card* **by** *simp*
have *greatest-eq-minus-1*: (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) = −*1*
using *a-eq-minus-1 greatest-eq-card to-nat-plus-one-less-card* **unfolding** *nrows-def* **by** *fastforce*
have *rref-suc*: *reduced-row-echelon-form-upt-k A* (*Suc k*)
proof (*rule rref-suc-if-all-rows-not-zero*)
show *reduced-row-echelon-form-upt-k A k* **using** *rref* **.**
show ∀ *n*. ¬ *is-zero-row-upt-k n k A* **using** *Greatest-is-minus-1 greatest-eq-minus-1 greatest-ge-nonzero-row*$'$[*OF rref* -] *not-zero-m* **by** *metis*
qed
show *?thesis* **using** *rref-upt-condition4*[*OF rref-suc*] **using** *not-zero-i-suc-k i-not-j* **unfolding** *B-eq-A i-not-j* **by** *blast*
qed

lemma *condition-4-part-5*:
fixes A::$'a$::{*field*}$^{\prime}$*columns*::{*mod-type*}$^{\prime}$*rows*::{*mod-type*} **and** k::*nat*
defines *ia*:*ia*≡(*if* ∀ *m*. *is-zero-row-upt-k m k A then 0 else to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1*)
defines *B*:*B*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *rref*: *reduced-row-echelon-form-upt-k A k*
and *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
and *i-not-j*: *i* ≠ *j*
and *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
and *greatest-not-card*: *Suc* (*to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*)) ≠ *nrows A*
and *greatest-less-ma*: (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1* ≤ *ma*
and *A-ma-k-not-zero*: *A* $ *ma* $ *from-nat k* ≠ *0*
shows *Gauss-Jordan-in-ij A* ((*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *j* $
(*LEAST n*. *Gauss-Jordan-in-ij A* ((*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *i* $ *n* ≠ *0*) = *0*
proof −
have *ia2*: *ia*=*to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1* **unfolding**

ia **using** *not-zero-m* **by** *presburger*
have *B-eq-Gauss*: $B = Gauss\text{-}Jordan\text{-}in\text{-}ij\ A\ ((GREATEST\ n.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ n\ k\ A) + 1)\ (from\text{-}nat\ k)$
unfolding *B Gauss-Jordan-column-k-def*
unfolding *ia2 Let-def fst-conv snd-conv*
using *greatest-not-card greatest-less-ma A-ma-k-not-zero*
by (*auto simp add*: *from-nat-to-nat-greatest*)
have *suc-greatest-not-zero*: $(GREATEST\ n.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ n\ k\ A) + 1 \neq 0$
using *Suc-eq-plus1 suc-not-zero greatest-not-card* **unfolding** *nrows-def* **by** *auto*
show *?thesis*
proof (*cases is-zero-row-upt-k i k A*)
case *True*
have *zero-i-k-B*: *is-zero-row-upt-k i k B* **unfolding** *B-eq-Gauss* **by** (*rule is-zero-after-Gauss*[*OF True not-zero-m rref greatest-less-ma A-ma-k-not-zero*])
hence *Gauss-Jordan-i-not-0*: $Gauss\text{-}Jordan\text{-}in\text{-}ij\ A\ ((GREATEST\ n.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ n\ k\ A) + 1)\ (from\text{-}nat\ k)\ \$\ (i)\ \$\ (from\text{-}nat\ k) \neq 0$
using *not-zero-i-suc-k* **unfolding** *B-eq-Gauss* **using** *is-zero-row-upt-k-suc* **by** *blast*
have *i-eq-greatest*: $i = ((GREATEST\ n.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ n\ k\ A) + 1)$
proof (*rule ccontr*)
assume *i-not-greatest*: $i \neq (GREATEST\ n.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ n\ k\ A) + 1$
have $Gauss\text{-}Jordan\text{-}in\text{-}ij\ A\ ((GREATEST\ n.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ n\ k\ A) + 1)\ (from\text{-}nat\ k)\ \$\ i\ \$\ (from\text{-}nat\ k) = 0$
proof (*rule Gauss-Jordan-in-ij-0*)
show $\exists\, n.\ A\ \$\ n\ \$\ from\text{-}nat\ k \neq 0 \wedge (GREATEST\ n.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ n\ k\ A) + 1 \leq n$ **using** *greatest-less-ma A-ma-k-not-zero* **by** *blast*
show $i \neq (GREATEST\ n.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ n\ k\ A) + 1$ **using** *i-not-greatest*
.
qed
thus *False* **using** *Gauss-Jordan-i-not-0* **by** *contradiction*
qed
have $Gauss\text{-}Jordan\text{-}i\text{-}1: Gauss\text{-}Jordan\text{-}in\text{-}ij\ A\ ((GREATEST\ n.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ n\ k\ A) + 1)\ (from\text{-}nat\ k)\ \$\ i\ \$\ (from\text{-}nat\ k) = 1$
unfolding *i-eq-greatest* **using** *Gauss-Jordan-in-ij-1 greatest-less-ma A-ma-k-not-zero* **by** *blast*
have $Least\text{-}eq\text{-}k: (LEAST\ ka.\ Gauss\text{-}Jordan\text{-}in\text{-}ij\ A\ ((GREATEST\ n.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ n\ k\ A) + 1)\ (from\text{-}nat\ k)\ \$\ i\ \$\ ka \neq 0) = from\text{-}nat\ k$
proof (*rule Least-equality*)
show $Gauss\text{-}Jordan\text{-}in\text{-}ij\ A\ ((GREATEST\ n.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ n\ k\ A) + 1)\ (from\text{-}nat\ k)\ \$\ i\ \$\ from\text{-}nat\ k \neq 0$ **using** *Gauss-Jordan-i-not-0* **.**
fix *y* **assume** $Gauss\text{-}Jordan\text{-}in\text{-}ij\ A\ ((GREATEST\ n.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ n\ k\ A) + 1)\ (from\text{-}nat\ k)\ \$\ i\ \$\ y \neq 0$
thus $from\text{-}nat\ k \leq y$ **using** *zero-i-k-B* **unfolding** *i-eq-greatest B-eq-Gauss* **by** (*metis is-zero-row-upt-k-def not-less to-nat-le*)
qed
show *?thesis* **using** *A-ma-k-not-zero Gauss-Jordan-in-ij-0′ Least-eq-k greatest-less-ma i-eq-greatest i-not-j* **by** *force*
next
case *False*

obtain n **where** *Ain-not-0*: $A\ \$\ i\ \$\ n \neq 0$ **and** *j-le-k*: *to-nat* $n < k$ **using** *False* **unfolding** *is-zero-row-upt-k-def* **by** *auto*
have *least-le-k*: *to-nat* (*LEAST ka*. $A\ \$\ i\ \$\ ka \neq 0$) $< k$
by (*metis* (*lifting*, *mono-tags*) *Ain-not-0 neq-iff j-le-k less-trans not-less-Least to-nat-mono*)
have *Least-eq*: (*LEAST ka*. *Gauss-Jordan-in-ij A* ((*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $\ \$\ i\ \$\ ka \neq 0$)
= (*LEAST ka*. $A\ \$\ i\ \$\ ka \neq 0$)
proof (*rule Least-equality*)
show *Gauss-Jordan-in-ij A* ((*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $\ \$\ i\ \$$ (*LEAST ka*. $A\ \$\ i\ \$\ ka \neq 0$) $\neq 0$
using *Gauss-Jordan-in-ij-preserves-previous-elements*[*OF rref False - suc-greatest-not-zero least-le-k*] **using** *greatest-less-ma A-ma-k-not-zero*
using *rref-upt-condition2*[*OF rref*] *False* **by** *fastforce*
fix *y* **assume** *Gauss-Jordan-y*:*Gauss-Jordan-in-ij A* ((*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $\ \$\ i\ \$\ y \neq 0$
show (*LEAST ka*. $A\ \$\ i\ \$\ ka \neq 0$) $\leq y$
proof (*cases to-nat y* $< k$)
case *False* **show** *?thesis* **by** (*metis* (*mono-tags*) *False least-le-k less-trans not-le-imp-less to-nat-from-nat to-nat-le*)
next
case *True*
have $A\ \$\ i\ \$\ y \neq 0$
using *Gauss-Jordan-y* **using** *Gauss-Jordan-in-ij-preserves-previous-elements*[*OF rref not-zero-m - suc-greatest-not-zero True*]
using *A-ma-k-not-zero greatest-less-ma* **by** *fastforce*
thus *?thesis* **by** (*rule Least-le*)
qed
qed
have *Gauss-Jordan-eq-A*: *Gauss-Jordan-in-ij A* ((*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $\ \$\ j\ \$$ (*LEAST n*. $A\ \$\ i\ \$\ n \neq 0$) =
$A\ \$\ j\ \$$ (*LEAST n*. $A\ \$\ i\ \$\ n \neq 0$)
using *Gauss-Jordan-in-ij-preserves-previous-elements*[*OF rref not-zero-m - suc-greatest-not-zero least-le-k*]
using *A-ma-k-not-zero greatest-less-ma* **by** *fastforce*
show *?thesis* **unfolding** *Least-eq* **using** *rref-upt-condition4*[*OF rref*]
using *False Gauss-Jordan-eq-A i-not-j* **by** *presburger*
qed
qed

lemma *condition-4*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*if* ∀ *m*. *is-zero-row-upt-k m k A then 0 else to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1*)
defines *B*:*B*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *rref*: *reduced-row-echelon-form-upt-k A k*
and *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
and *i-not-j*: $i \neq j$

shows $B \$ j \$ (LEAST\ n.\ B \$ i \$ n \neq 0) = 0$
proof (*unfold B Gauss-Jordan-column-k-def ia Let-def fst-conv snd-conv*, *auto*, *unfold from-nat-to-nat-greatest from-nat-0*)
assume *all-zero*: $\forall m.$ *is-zero-row-upt-k m k A*
and *all-zero-k*: $\forall m \geq 0.\ A \$ m \$$ *from-nat k* $= 0$
show $A \$ j \$ (LEAST\ n.\ A \$ i \$ n \neq 0) = 0$ **using** *condition-4-part-1*[*OF - all-zero*] **using** *all-zero-k not-zero-i-suc-k least-mod-type* **unfolding** *B ia* **by** *blast*
next
fix *m*
assume *all-zero*: $\forall m.$ *is-zero-row-upt-k m k A*
and *Amk-not-zero*: $A \$ m \$$ *from-nat k* $\neq 0$
show *Gauss-Jordan-in-ij A 0* (*from-nat k*) $\$ j \$ (LEAST\ n.$ *Gauss-Jordan-in-ij A 0* (*from-nat k*) $\$ i \$ n \neq 0) = 0$
using *condition-4-part-2*[*OF - i-not-j all-zero Amk-not-zero*] **using** *not-zero-i-suc-k* **unfolding** *B ia* **.**
next
fix *m* **assume** *not-zero-m*: $\neg$ *is-zero-row-upt-k m k A*
and *zero-below-greatest*: $\forall m \geq (GREATEST\ n.\ \neg$ *is-zero-row-upt-k n k A*$) + 1.$ $A \$ m \$$ *from-nat k* $= 0$
show $A \$ j \$ (LEAST\ n.\ A \$ i \$ n \neq 0) = 0$
using *condition-4-part-3*[*OF rref - i-not-j not-zero-m zero-below-greatest*] **using** *not-zero-i-suc-k* **unfolding** *B ia* **.**
next
fix *m*
assume *not-zero-m*: $\neg$ *is-zero-row-upt-k m k A*
and *greatest-eq-card*: *Suc* (*to-nat* $(GREATEST\ n.\ \neg$ *is-zero-row-upt-k n k A*$))$ $=$ *nrows A*
show $A \$ j \$ (LEAST\ n.\ A \$ i \$ n \neq 0) = 0$
using *condition-4-part-4*[*OF rref - i-not-j not-zero-m greatest-eq-card*] **using** *not-zero-i-suc-k* **unfolding** *B ia* **.**
next
fix *m ma*
assume *not-zero-m*: $\neg$ *is-zero-row-upt-k m k A*
and *greatest-not-card*: *Suc* (*to-nat* $(GREATEST\ n.\ \neg$ *is-zero-row-upt-k n k A*$))$ $\neq$ *nrows A*
and *greatest-less-ma*: $(GREATEST\ n.\ \neg$ *is-zero-row-upt-k n k A*$) + 1 \leq$ *ma*
and *A-ma-k-not-zero*: $A \$ ma \$$ *from-nat k* $\neq 0$
show *Gauss-Jordan-in-ij A* $((GREATEST\ n.\ \neg$ *is-zero-row-upt-k n k A*$) + 1)$ (*from-nat k*) $\$ j \$$
$(LEAST\ n.$ *Gauss-Jordan-in-ij A* $((GREATEST\ n.\ \neg$ *is-zero-row-upt-k n k A*$)$ $+ 1)$ (*from-nat k*) $\$ i \$ n \neq 0) = 0$
using *condition-4-part-5*[*OF rref - i-not-j not-zero-m greatest-not-card greatest-less-ma A-ma-k-not-zero*] **using** *not-zero-i-suc-k* **unfolding** *B ia* **.**
qed

lemma *reduced-row-echelon-form-upt-k-Gauss-Jordan-column-k*:
fixes *A*::*'a*::{*field*}^*columns*::{*mod-type*}^*rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*if* $\forall m.$ *is-zero-row-upt-k m k A then 0 else to-nat* $(GREATEST$

n. ¬ *is-zero-row-upt-k n k A*) + *1*)
defines *B*:*B*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *rref*: *reduced-row-echelon-form-upt-k A k*
shows *reduced-row-echelon-form-upt-k B* (*Suc k*)
proof (*rule reduced-row-echelon-form-upt-k-intro*, *auto*)
show ⋀*i j*. *is-zero-row-upt-k i* (*Suc k*) *B* ⟹ *i* < *j* ⟹ *is-zero-row-upt-k j* (*Suc k*) *B* **using** *condition-1 assms* **by** *blast*
show ⋀*i*. ¬ *is-zero-row-upt-k i* (*Suc k*) *B* ⟹ *B* $ *i* $ (*LEAST k*. *B* $ *i* $ *k* ≠ *0*) = *1* **using** *condition-2 assms* **by** *blast*
show ⋀*i*. *i* < *i* + *1* ⟹ ¬ *is-zero-row-upt-k i* (*Suc k*) *B* ⟹ ¬ *is-zero-row-upt-k* (*i* + *1*) (*Suc k*) *B* ⟹ (*LEAST n*. *B* $ *i* $ *n* ≠ *0*) < (*LEAST n*. *B* $ (*i* + *1*) $ *n* ≠ *0*) **using** *condition-3 assms* **by** *blast*
show ⋀*i j*. ¬ *is-zero-row-upt-k i* (*Suc k*) *B* ⟹ *i* ≠ *j* ⟹ *B* $ *j* $ (*LEAST n*. *B* $ *i* $ *n* ≠ *0*) = *0* **using** *condition-4 assms* **by** *blast*
qed

lemma *foldl-Gauss-condition-1*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
assumes ∀ *m*. *is-zero-row-upt-k m k A*
and ∀ *m*≥*0*. *A* $ *m* $ *from-nat k* = *0*
shows *is-zero-row-upt-k m* (*Suc k*) *A*
by (*rule is-zero-row-upt-k-suc*, *auto simp add*: *assms least-mod-type*)

lemma *foldl-Gauss-condition-2*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
assumes *k*: *k* < *ncols A*
and *all-zero*: ∀ *m*. *is-zero-row-upt-k m k A*
and *Amk-not-zero*: *A* $ *m* $ *from-nat k* ≠ *0*
shows ∃ *m*. ¬ *is-zero-row-upt-k m* (*Suc k*) (*Gauss-Jordan-in-ij A 0* (*from-nat k*))
proof −
have *to-nat-from-nat-k-suc*: *to-nat* (*from-nat k*::*'columns*) < (*Suc k*) **using** *to-nat-from-nat-id*[*OF k*[*unfolded ncols-def*]] **by** *simp*
have *A0k-eq-1*: (*Gauss-Jordan-in-ij A 0* (*from-nat k*)) $ *0* $ (*from-nat k*) = *1*
by (*rule Gauss-Jordan-in-ij-1*, *auto intro*!: *Amk-not-zero least-mod-type*)
have ¬ *is-zero-row-upt-k 0* (*Suc k*) (*Gauss-Jordan-in-ij A 0* (*from-nat k*))
unfolding *is-zero-row-upt-k-def*
using *A0k-eq-1 to-nat-from-nat-k-suc* **by** *force*
thus *?thesis* **by** *blast*
qed

lemma *foldl-Gauss-condition-3*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
assumes *k*: *k* < *ncols A*
and *all-zero*: ∀ *m*. *is-zero-row-upt-k m k A*
and *Amk-not-zero*: *A* $ *m* $ *from-nat k* ≠ *0*
and ¬ *is-zero-row-upt-k ma* (*Suc k*) (*Gauss-Jordan-in-ij A 0* (*from-nat k*))

shows *to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n* (*Suc k*) (*Gauss-Jordan-in-ij A 0* (*from-nat k*))) = *0*
proof (*unfold to-nat-eq-0*, *rule Greatest-equality*)
have *to-nat-from-nat-k-suc*: *to-nat* (*from-nat k*::*'columns*) < *Suc* (*k*) **using** *to-nat-from-nat-id*[*OF k*[*unfolded ncols-def*]] **by** *simp*
have *A0k-eq-1*: (*Gauss-Jordan-in-ij A 0* (*from-nat k*)) $ *0* $ (*from-nat k*) = *1*
by (*rule Gauss-Jordan-in-ij-1*, *auto intro*!: *Amk-not-zero least-mod-type*)
show ¬ *is-zero-row-upt-k 0* (*Suc k*) (*Gauss-Jordan-in-ij A 0* (*from-nat k*))
unfolding *is-zero-row-upt-k-def*
using *A0k-eq-1 to-nat-from-nat-k-suc* **by** *force*
fix *y*
assume *not-zero-y*: ¬ *is-zero-row-upt-k y* (*Suc k*) (*Gauss-Jordan-in-ij A 0* (*from-nat k*))
have *y-eq-0*: *y*=*0*
proof (*rule ccontr*)
assume *y-not-0*: *y* ≠ *0*
have *is-zero-row-upt-k y* (*Suc k*) (*Gauss-Jordan-in-ij A 0* (*from-nat k*)) **unfolding** *is-zero-row-upt-k-def*
proof (*clarify*)
fix *j*::*'columns* **assume** *j*: *to-nat j* < *Suc k*
show *Gauss-Jordan-in-ij A 0* (*from-nat k*) $ *y* $ *j* = *0*
proof (*cases to-nat j* = *k*)
case *True* **show** *?thesis* **unfolding** *to-nat-from-nat*[*OF True*]
by (*rule Gauss-Jordan-in-ij-0*[*OF - y-not-0*], *unfold to-nat-from-nat*[*OF True*, *symmetric*], *auto intro*!: *y-not-0 least-mod-type Amk-not-zero*)
next
case *False* **hence** *j-less-k*: *to-nat j* < *k* **by** (*metis j less-SucE*)
show *?thesis* **using** *Gauss-Jordan-in-ij-preserves-previous-elements'*[*OF all-zero j-less-k Amk-not-zero*]
using *all-zero j-less-k* **unfolding** *is-zero-row-upt-k-def* **by** *presburger*
qed
qed
thus *False* **using** *not-zero-y* **by** *contradiction*
qed
thus *y*≤*0* **using** *least-mod-type* **by** *simp*
qed

lemma *foldl-Gauss-condition-5*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
assumes *rref-A*: *reduced-row-echelon-form-upt-k A k*
and *not-zero-a*:¬ *is-zero-row-upt-k a k A*
and *all-zero-below-greatest*: ∀ *m*≥(*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1*. *A* $ *m* $ *from-nat k* = *0*
shows (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) = (*GREATEST n*. ¬ *is-zero-row-upt-k n* (*Suc k*) *A*)
proof −
have ⋀*n*. (*is-zero-row-upt-k n* (*Suc k*) *A*) = (*is-zero-row-upt-k n k A*)
proof

fix n **assume** *is-zero-row-upt-k n (Suc k) A*
thus *is-zero-row-upt-k n k A* **using** *is-zero-row-upt-k-le* **by** *fast*
next
fix n **assume** *zero-n-k*: *is-zero-row-upt-k n k A*
have $n>$(*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) **by** (*rule greatest-less-zero-row*[*OF rref-A zero-n-k*], *auto intro*!: *not-zero-a*)
hence *n-ge-gratest*: $n \geq$ (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1* **using** *le-Suc* **by** *blast*
hence *A-nk-zero*: *A $ n $ (from-nat k) = 0* **using** *all-zero-below-greatest* **by** *fast*
show *is-zero-row-upt-k n (Suc k) A* **by** (*rule is-zero-row-upt-k-suc*[*OF zero-n-k A-nk-zero*])
qed
thus (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) = (*GREATEST n.* ¬ *is-zero-row-upt-k n (Suc k) A*) **by** *simp*
qed

lemma *foldl-Gauss-condition-6*:
fixes *A*::*'a*::{*field*}^*columns*::{*mod-type*}^*rows*::{*mod-type*} **and** *k*::*nat*
assumes *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
and *eq-card*: *Suc* (*to-nat* (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*)) = *nrows A*
shows *nrows A* = *Suc* (*to-nat* (*GREATEST n.* ¬ *is-zero-row-upt-k n* (*Suc k*) *A*))
proof −
have (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1* = *0* **using** *greatest-plus-one-eq-0*[*OF eq-card*] **.**
hence *greatest-k-eq-minus-1*: (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) = −*1* **using** *a-eq-minus-1* **by** *blast*
have (*GREATEST n.* ¬ *is-zero-row-upt-k n* (*Suc k*) *A*) = −*1*
proof (*rule Greatest-equality*)
show ¬ *is-zero-row-upt-k* (− *1*) (*Suc k*) *A*
using *GreatestI-ex greatest-k-eq-minus-1 is-zero-row-upt-k-le not-zero-m* **by** *force*
show ⋀*y.* ¬ *is-zero-row-upt-k y* (*Suc k*) *A* ⟹ *y* ≤ −*1* **using** *Greatest-is-minus-1* **by** *fast*
qed
thus *nrows A* = *Suc* (*to-nat* (*GREATEST n.* ¬ *is-zero-row-upt-k n* (*Suc k*) *A*)) **using** *eq-card greatest-k-eq-minus-1* **by** *fastforce*
qed

lemma *foldl-Gauss-condition-8*:
fixes *A*::*'a*::{*field*}^*columns*::{*mod-type*}^*rows*::{*mod-type*} **and** *k*::*nat*
assumes *k*: *k* < *ncols A*
and *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
and *A-ma-k*: *A $ ma $ from-nat k* ≠ *0*
and *ma*: (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1* ≤ *ma*

shows $\exists m.\ \neg$ *is-zero-row-upt-k m* (*Suc k*) (*Gauss-Jordan-in-ij A* ((*GREATEST n.* $\neg$ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*))
proof −
define *Greatest-plus-one* **where** *Greatest-plus-one* = (*GREATEST n.* $\neg$ *is-zero-row-upt-k n k A*) + *1*
have *to-nat-from-nat-k-suc*: *to-nat* (*from-nat k*::*'columns*) < (*Suc k*) **using** *to-nat-from-nat-id*[*OF k*[*unfolded ncols-def*]] **by** *simp*
have *Gauss-eq-1*: (*Gauss-Jordan-in-ij A Greatest-plus-one* (*from-nat k*)) $ *Greatest-plus-one* $ (*from-nat k*) = *1*
by (*unfold Greatest-plus-one-def*, *rule Gauss-Jordan-in-ij-1*, *auto intro*!: *A-ma-k ma*)
show $\exists m.\ \neg$ *is-zero-row-upt-k m* (*Suc k*) (*Gauss-Jordan-in-ij A* (*Greatest-plus-one*) (*from-nat k*))
by (*rule exI*[*of - Greatest-plus-one*], *unfold is-zero-row-upt-k-def*, *auto*, *rule exI*[*of - from-nat k*], *simp add*: *Gauss-eq-1 to-nat-from-nat-k-suc*)
qed

lemma *foldl-Gauss-condition-9*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
assumes *k*: *k* < *ncols A*
and *rref-A*: *reduced-row-echelon-form-upt-k A k*
assumes *not-zero-m*: $\neg$ *is-zero-row-upt-k m k A*
and *suc-greatest-not-card*: *Suc* (*to-nat* (*GREATEST n.* $\neg$ *is-zero-row-upt-k n k A*)) ≠ *nrows A*
and *greatest-less-ma*: (*GREATEST n.* $\neg$ *is-zero-row-upt-k n k A*) + *1* ≤ *ma*
and *A-ma-k*: *A* $ *ma* $ *from-nat k* ≠ *0*
shows *Suc* (*to-nat* (*GREATEST n.* $\neg$ *is-zero-row-upt-k n k A*)) =
to-nat(*GREATEST n.* $\neg$ *is-zero-row-upt-k n* (*Suc k*) (*Gauss-Jordan-in-ij A* ((*GREATEST n.* $\neg$ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*)))
proof −
define *Greatest-plus-one* **where** *Greatest-plus-one* = (*GREATEST n.* $\neg$ *is-zero-row-upt-k n k A*) + *1*
have *to-nat-from-nat-k-suc*: *to-nat* (*from-nat k*::*'columns*) < (*Suc k*) **using** *to-nat-from-nat-id*[*OF k*[*unfolded ncols-def*]] **by** *simp*
have *greatest-plus-one-not-zero*: *Greatest-plus-one* ≠ *0*
proof −
have *to-nat* (*GREATEST n.* $\neg$ *is-zero-row-upt-k n k A*) < *nrows A* **using** *to-nat-less-card* **unfolding** *nrows-def* **by** *blast*
hence *to-nat* (*GREATEST n.* $\neg$ *is-zero-row-upt-k n k A*) + *1* < *nrows A* **using** *suc-greatest-not-card* **by** *linarith*
show *?thesis* **unfolding** *Greatest-plus-one-def* **by** (*rule suc-not-zero*[*OF suc-greatest-not-card*[*unfolded Suc-eq-plus1 nrows-def*]])
qed
have *greatest-eq*: *Greatest-plus-one* = (*GREATEST n.* $\neg$ *is-zero-row-upt-k n* (*Suc k*) (*Gauss-Jordan-in-ij A Greatest-plus-one* (*from-nat k*)))
proof (*rule Greatest-equality*[*symmetric*])
have (*Gauss-Jordan-in-ij A Greatest-plus-one* (*from-nat k*)) $ (*Greatest-plus-one*) $ (*from-nat k*) = *1*
by (*unfold Greatest-plus-one-def*, *rule Gauss-Jordan-in-ij-1*, *auto intro*!: *great-*

est-less-ma A-ma-k)
thus ¬ *is-zero-row-upt-k Greatest-plus-one* (*Suc k*) (*Gauss-Jordan-in-ij A Greatest-plus-one* (*from-nat k*))
using *to-nat-from-nat-k-suc*
unfolding *is-zero-row-upt-k-def* **by** *fastforce*
fix *y*
assume *not-zero-y*: ¬ *is-zero-row-upt-k y* (*Suc k*) (*Gauss-Jordan-in-ij A Greatest-plus-one* (*from-nat k*))
show *y* ≤ *Greatest-plus-one*
proof (*cases y*<*Greatest-plus-one*)
case *True* **thus** *?thesis* **by** *simp*
next
case *False* **hence** *y-ge-greatest*: *y*≥*Greatest-plus-one* **by** *simp*
have *y*=*Greatest-plus-one*
proof (*rule ccontr*)
assume *y-not-greatest*: *y* ≠ *Greatest-plus-one*
have (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) < *y* **using** *greatest-plus-one-not-zero*
using *Suc-le'* *less-le-trans y-ge-greatest* **unfolding** *Greatest-plus-one-def* **by** *auto*
hence *zero-row-y-upt-k*: *is-zero-row-upt-k y k A* **using** *not-greater-Greatest*[*of* λ*n*. ¬ *is-zero-row-upt-k n k A y*] **unfolding** *Greatest-plus-one-def* **by** *fast*
have *is-zero-row-upt-k y* (*Suc k*) (*Gauss-Jordan-in-ij A Greatest-plus-one* (*from-nat k*)) **unfolding** *is-zero-row-upt-k-def*
proof (*clarify*)
fix *j*::*'columns* **assume** *j*: *to-nat j* < *Suc k*
show *Gauss-Jordan-in-ij A Greatest-plus-one* (*from-nat k*) $ *y* $ *j* = *0*
proof (*cases j*=*from-nat k*)
case *True*
show *?thesis*
proof (*unfold True*, *rule Gauss-Jordan-in-ij-0*[*OF - y-not-greatest*], *rule exI*[*of - ma*], *rule conjI*)
show *A* $ *ma* $ *from-nat k* ≠ *0* **using** *A-ma-k* **.**
show *Greatest-plus-one* ≤ *ma* **using** *greatest-less-ma* **unfolding** *Greatest-plus-one-def* **.**
qed
next
case *False* **hence** *j-le-suc-k*: *to-nat j* < *Suc k* **using** *j* **by** *simp*
have *Gauss-Jordan-in-ij A Greatest-plus-one* (*from-nat k*) $ *y* $ *j* = *A* $ *y* $ *j* **unfolding** *Greatest-plus-one-def*
proof (*rule Gauss-Jordan-in-ij-preserves-previous-elements*)
show *reduced-row-echelon-form-upt-k A k* **using** *rref-A* **.**
show ¬ *is-zero-row-upt-k m k A* **using** *not-zero-m* **.**
show ∃ *n*. *A* $ *n* $ *from-nat k* ≠ *0* ∧ (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1* ≤ *n* **using** *A-ma-k greatest-less-ma* **by** *blast*
show (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1* ≠ *0* **using** *greatest-plus-one-not-zero* **unfolding** *Greatest-plus-one-def* **.**
show *to-nat j* < *k* **using** *False from-nat-to-nat-id j-le-suc-k less-antisym* **by** *fastforce*

```
        qed
      also have ... = 0 using zero-row-y-upt-k unfolding is-zero-row-upt-k-def
        using False le-imp-less-or-eq from-nat-to-nat-id j-le-suc-k less-Suc-eq-le
by fastforce
      finally show Gauss-Jordan-in-ij A Greatest-plus-one (from-nat k) $ y $
j = 0 .
    qed
   qed
   thus False using not-zero-y by contradiction
  qed
  thus y ≤ Greatest-plus-one using y-ge-greatest by blast
 qed
qed
show Suc (to-nat (GREATEST n. ¬ is-zero-row-upt-k n k A)) =
  to-nat (GREATEST n. ¬ is-zero-row-upt-k n (Suc k) (Gauss-Jordan-in-ij A
((GREATEST n. ¬ is-zero-row-upt-k n k A) + 1) (from-nat k)))
  unfolding greatest-eq[unfolded Greatest-plus-one-def, symmetric]
  unfolding add-to-nat-def
  unfolding to-nat-1
  using to-nat-from-nat-id to-nat-plus-one-less-card
  using greatest-plus-one-not-zero[unfolded Greatest-plus-one-def]
  by force
qed
```

The following lemma is one of most important ones in the verification of the Gauss-Jordan algorithm. The aim is to prove two statements about *Gauss-Jordan-upt-k ?A ?k = snd (foldl Gauss-Jordan-column-k (0, ?A) [0..<Suc ?k])* (one about the result is on rref and another about the index). The reason of doing that way is because both statements need them mutually to be proved. As the proof is made using induction, two base cases and two induction steps appear.

```
lemma rref-and-index-Gauss-Jordan-upt-k:
 fixes A::'a::{field}^'columns::{mod-type}^'rows::{mod-type} and k::nat
 assumes k < ncols A
 shows rref-Gauss-Jordan-upt-k: reduced-row-echelon-form-upt-k (Gauss-Jordan-upt-k
A k) (Suc k)
 and snd-Gauss-Jordan-upt-k:
 foldl Gauss-Jordan-column-k (0, A) [0..<Suc k] =
 (if ∀ m. is-zero-row-upt-k m (Suc k) (snd (foldl Gauss-Jordan-column-k (0, A)
[0..<Suc k])) then 0
 else to-nat (GREATEST n. ¬ is-zero-row-upt-k n (Suc k) (snd (foldl Gauss-Jordan-column-k
(0, A) [0..<Suc k]))) + 1,
 snd (foldl Gauss-Jordan-column-k (0, A) [0..<Suc k]))
 using assms
proof (induct k)
  — Two base cases, one for each show
  — The first one
 show reduced-row-echelon-form-upt-k (Gauss-Jordan-upt-k A 0) (Suc 0)
```

unfolding *Gauss-Jordan-upt-k-def* **apply** *auto*
using *reduced-row-echelon-form-upt-k-Gauss-Jordan-column-k*[*OF rref-upt-0*, *of A*] **using** *is-zero-row-utp-0'*[*of A*] **by** *simp*
— The second base case
have *rw-upt*: [*0..<Suc 0*] = [*0*] **by** *simp*
show *foldl Gauss-Jordan-column-k* (*0*, *A*) [*0..<Suc 0*] =
(*if* ∀ *m*. *is-zero-row-upt-k m* (*Suc 0*) (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0..<Suc 0*])) *then 0*
else to-nat (*GREATEST n*. ¬ *is-zero-row-upt-k n* (*Suc 0*) (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0..<Suc 0*]))) + *1*,
snd (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0..<Suc 0*]))
unfolding *rw-upt*
unfolding *foldl.simps*
unfolding *Gauss-Jordan-column-k-def Let-def from-nat-0 fst-conv snd-conv*
unfolding *is-zero-row-upt-k-def*
apply (*auto simp add*: *least-mod-type to-nat-eq-0*)
apply (*metis Gauss-Jordan-in-ij-1 least-mod-type zero-neq-one*)
by (*metis* (*lifting*, *mono-tags*) *Gauss-Jordan-in-ij-0 GreatestI-ex least-mod-type*)
next
— Now we begin with the proof of the induction step of the first show. We will make use the induction hypothesis of the second show
fix *k*
assume (*k* < *ncols A* ⟹ *reduced-row-echelon-form-upt-k* (*Gauss-Jordan-upt-k A k*) (*Suc k*))
and (*k* < *ncols A* ⟹
foldl Gauss-Jordan-column-k (*0*, *A*) [*0..<Suc k*] =
(*if* ∀ *m*. *is-zero-row-upt-k m* (*Suc k*) (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0..<Suc k*])) *then 0*
else to-nat (*GREATEST n*. ¬ *is-zero-row-upt-k n* (*Suc k*) (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0..<Suc k*]))) + *1*,
snd (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0..<Suc k*])))
and *k*: *Suc k* < *ncols A*
hence *hyp-rref*: *reduced-row-echelon-form-upt-k* (*Gauss-Jordan-upt-k A k*) (*Suc k*)
and *hyp-foldl*: *foldl Gauss-Jordan-column-k* (*0*, *A*) [*0..<Suc k*] =
(*if* ∀ *m*. *is-zero-row-upt-k m* (*Suc k*) (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0..<Suc k*])) *then 0*
else to-nat (*GREATEST n*. ¬ *is-zero-row-upt-k n* (*Suc k*) (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0..<Suc k*]))) + *1*,
snd (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0..<Suc k*]))
by *simp*+
have *rw*: [*0..<Suc* (*Suc k*)]= [*0..<*(*Suc k*)] @ [(*Suc k*)] **by** *auto*
have *rw2*: (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0..<Suc k*]) =
(*if* ∀ *m*. *is-zero-row-upt-k m* (*Suc k*) (*Gauss-Jordan-upt-k A k*) *then 0 else to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n* (*Suc k*) (*Gauss-Jordan-upt-k A k*)) + *1*,
Gauss-Jordan-upt-k A k) **unfolding** *Gauss-Jordan-upt-k-def* **using** *hyp-foldl* **by** *fast*
show *reduced-row-echelon-form-upt-k* (*Gauss-Jordan-upt-k A* (*Suc k*)) (*Suc* (*Suc k*))

unfolding *Gauss-Jordan-upt-k-def* **unfolding** *rw* **unfolding** *foldl-append* **unfolding** *foldl.simps* **unfolding** *rw2*

by (*rule reduced-row-echelon-form-upt-k-Gauss-Jordan-column-k*[*OF hyp-rref*])

— Making use of the same hypotheses of above proof, we begin with the proof of the induction step of the second show.

have *fst-foldl*: *fst* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc k*]) =

fst (*if* ∀ *m*. *is-zero-row-upt-k m* (*Suc k*) (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc k*])) *then 0*

else to-nat (*GREATEST n*. ¬ *is-zero-row-upt-k n* (*Suc k*) (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc k*]))) + *1*,

snd (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc k*])) **using** *hyp-foldl* **by** *simp*

show *foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc* (*Suc k*)] =

(*if* ∀ *m*. *is-zero-row-upt-k m* (*Suc* (*Suc k*)) (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc* (*Suc k*)])) *then 0*

else to-nat (*GREATEST n*. ¬ *is-zero-row-upt-k n* (*Suc* (*Suc k*)) (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc* (*Suc k*)]))) + *1*,

snd (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc* (*Suc k*)]))

proof (*rule prod-eqI*)

show *snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc* (*Suc k*)]) =

snd (*if* ∀ *m*. *is-zero-row-upt-k m* (*Suc* (*Suc k*)) (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc* (*Suc k*)])) *then 0*

else to-nat (*GREATEST n*. ¬ *is-zero-row-upt-k n* (*Suc* (*Suc k*)) (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc* (*Suc k*)]))) + *1*,

snd (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc* (*Suc k*)]))

unfolding *Gauss-Jordan-upt-k-def* **by** *force*

define *A′* **where** *A′* = *snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc k*])

have *ncols-eq*: *ncols A* = *ncols A′* **unfolding** *A′-def ncols-def* **..**

have *rref-A′*: *reduced-row-echelon-form-upt-k A′* (*Suc k*) **using** *hyp-rref* **unfolding** *A′-def Gauss-Jordan-upt-k-def* **.**

show *fst* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc* (*Suc k*)]) =

fst (*if* ∀ *m*. *is-zero-row-upt-k m* (*Suc* (*Suc k*)) (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc* (*Suc k*)])) *then 0*

else to-nat (*GREATEST n*. ¬ *is-zero-row-upt-k n* (*Suc* (*Suc k*)) (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc* (*Suc k*)]))) + *1*,

snd (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc* (*Suc k*)]))

apply (*simp only*: *rw*)

apply (*simp only*: *foldl-append*)

apply (*simp only*: *foldl.simps*)

apply (*simp only*: *Gauss-Jordan-column-k-def Let-def fst-foldl*)

apply (*simp only*: *A′-def*[*symmetric*])

apply *auto*

apply (*simp-all only*: *from-nat-0 from-nat-to-nat-greatest*)

proof −

fix *m* **assume** ∀ *m*. *is-zero-row-upt-k m* (*Suc k*) *A′* **and** ∀ *m*≥*0*. *A′* $ *m* $ *from-nat* (*Suc k*) = *0*

thus *is-zero-row-upt-k m* (*Suc* (*Suc k*)) *A′* **using** *foldl-Gauss-condition-1* **by** *blast*

next

fix *m*

assume $\forall\, m.$ *is-zero-row-upt-k* m (*Suc* k) A'
and A' \$ m \$ *from-nat* (*Suc* k) $\neq$ *0*
thus $\exists\, m.\ \neg$ *is-zero-row-upt-k* m (*Suc* (*Suc* k)) (*Gauss-Jordan-in-ij* A' *0* (*from-nat* (*Suc* k)))
using *foldl-Gauss-condition-2* k *ncols-eq* **by** *simp*
next
fix m *ma*
assume $\forall\, m.$ *is-zero-row-upt-k* m (*Suc* k) A'
and A' \$ m \$ *from-nat* (*Suc* k) $\neq$ *0*
and $\neg$ *is-zero-row-upt-k ma* (*Suc* (*Suc* k)) (*Gauss-Jordan-in-ij* A' *0* (*from-nat* (*Suc* k)))
thus *to-nat* (*GREATEST* $n.\ \neg$ *is-zero-row-upt-k* n (*Suc* (*Suc* k)) (*Gauss-Jordan-in-ij* A' *0* (*from-nat* (*Suc* k)))) = *0*
using *foldl-Gauss-condition-3* k *ncols-eq* **by** *simp*
next
fix m **assume** $\neg$ *is-zero-row-upt-k* m (*Suc* k) A'
thus $\exists\, m.\ \neg$ *is-zero-row-upt-k* m (*Suc* (*Suc* k)) A' **and** $\exists\, m.\ \neg$ *is-zero-row-upt-k* m (*Suc* (*Suc* k)) A' **using** *is-zero-row-upt-k-le* **by** *blast+*
next
fix m
assume *not-zero-m*: $\neg$ *is-zero-row-upt-k* m (*Suc* k) A'
and *zero-below-greatest*: $\forall\, m{\geq}$(*GREATEST* $n.\ \neg$ *is-zero-row-upt-k* n (*Suc* k) A') + *1*. A' \$ m \$ *from-nat* (*Suc* k) = *0*
show (*GREATEST* $n.\ \neg$ *is-zero-row-upt-k* n (*Suc* k) A') = (*GREATEST* $n.\ \neg$ *is-zero-row-upt-k* n (*Suc* (*Suc* k)) A')
by (*rule foldl-Gauss-condition-5*[*OF rref-A' not-zero-m zero-below-greatest*])
next
fix m **assume** $\neg$ *is-zero-row-upt-k* m (*Suc* k) A' **and** *Suc* (*to-nat* (*GREATEST* $n.\ \neg$ *is-zero-row-upt-k* n (*Suc* k) A')) = *nrows* A'
thus *nrows* A' = *Suc* (*to-nat* (*GREATEST* $n.\ \neg$ *is-zero-row-upt-k* n (*Suc* (*Suc* k)) A'))
using *foldl-Gauss-condition-6* **by** *blast*
next
fix m *ma*
assume $\neg$ *is-zero-row-upt-k* m (*Suc* k) A'
and (*GREATEST* $n.\ \neg$ *is-zero-row-upt-k* n (*Suc* k) A') + *1* $\leq$ *ma*
and A' \$ *ma* \$ *from-nat* (*Suc* k) $\neq$ *0*
thus $\exists\, m.\ \neg$ *is-zero-row-upt-k* m (*Suc* (*Suc* k)) (*Gauss-Jordan-in-ij* A' ((*GREATEST* $n.\ \neg$ *is-zero-row-upt-k* n (*Suc* k) A') + *1*) (*from-nat* (*Suc* k)))
using *foldl-Gauss-condition-8* **using** k *ncols-eq* **by** *simp*
next
fix m *ma mb*
assume $\neg$ *is-zero-row-upt-k* m (*Suc* k) A' **and**
Suc (*to-nat* (*GREATEST* $n.\ \neg$ *is-zero-row-upt-k* n (*Suc* k) A')) $\neq$ *nrows* A'
and (*GREATEST* $n.\ \neg$ *is-zero-row-upt-k* n (*Suc* k) A') + *1* $\leq$ *ma*
and A' \$ *ma* \$ *from-nat* (*Suc* k) $\neq$ *0*
and $\neg$ *is-zero-row-upt-k mb* (*Suc* (*Suc* k)) (*Gauss-Jordan-in-ij* A' ((*GREATEST* $n.\ \neg$ *is-zero-row-upt-k* n (*Suc* k) A') + *1*) (*from-nat* (*Suc* k)))
thus *Suc* (*to-nat* (*GREATEST* $n.\ \neg$ *is-zero-row-upt-k* n (*Suc* k) A')) =

to-nat (*GREATEST n*. ¬ *is-zero-row-upt-k n* (*Suc* (*Suc k*)) (*Gauss-Jordan-in-ij* A' ((*GREATEST n*. ¬ *is-zero-row-upt-k n* (*Suc k*) A') + *1*) (*from-nat* (*Suc k*))))
using *foldl-Gauss-condition-9*[*OF k*[*unfolded ncols-eq*] *rref-A*'] **unfolding** *nrows-def* **by** *blast*
qed
qed
qed

corollary *rref-Gauss-Jordan*:
fixes *A*::'*a*::{*field*}^'*columns*::{*mod-type*}^'*rows*::{*mod-type*}
shows *reduced-row-echelon-form* (*Gauss-Jordan A*)
proof −
have *CARD*('*columns*) − *1* < *CARD*('*columns*) **by** *fastforce*
thus *reduced-row-echelon-form* (*Gauss-Jordan A*)
unfolding *reduced-row-echelon-form-def* **unfolding** *Gauss-Jordan-def*
using *rref-Gauss-Jordan-upt-k* **unfolding** *ncols-def*
by (*metis* (*mono-tags*) *diff-Suc-1 lessE*)
qed

lemma *independent-not-zero-rows-rref*:
fixes *A*::'*a*::{*field*}^'*m*::{*mod-type*}^'*n*::{*finite*,*one*,*plus*,*ord*}
assumes *rref-A*: *reduced-row-echelon-form A*
shows *vec.independent* {*row i A* |*i*. *row i A* ≠ *0*}
proof
define *R* **where** *R* = {*row i A* |*i*. *row i A* ≠ *0*}
assume *dep*: *vec.dependent R*
from *this* **obtain** *a* **where** *a-in-R*: *a*∈*R* **and** *a-in-span*: *a* ∈ *vec.span* (*R* − {*a*})
unfolding *vec.dependent-def* **by** *fast*
from *a-in-R* **obtain** *i* **where** *a-eq-row-i-A*: *a*=*row i A* **unfolding** *R-def* **by** *blast*
hence *a-eq-Ai*: *a* = *A* $ *i* **unfolding** *row-def* **unfolding** *vec-nth-inverse* **.**
have *row-i-A-not-zero*: ¬ *is-zero-row i A* **using** *a-in-R*
unfolding *R-def is-zero-row-def is-zero-row-upt-ncols row-def vec-nth-inverse*
unfolding *vec-lambda-unique zero-vec-def mem-Collect-eq* **using** *a-eq-Ai* **by** *force*
define *least-n* **where** *least-n* = (*LEAST n*. *A* $ *i* $ *n* ≠ *0*)
have *span-rw*: *vec.span* (*R* − {*a*}) = *range* (λ*u*. ∑ *v*∈*R* − {*a*}. *u v* ∗*s v*)
proof (*rule vec.span-finite*)
show *finite* (*R* − {*a*}) **using** *finite-rows*[*of A*] **unfolding** *rows-def R-def* **by** *simp*
qed
from *this* **obtain** *f* **where** *f*: (∑ *v*∈(*R* − {*a*}). *f v* ∗*s v*) = *a* **using** *a-in-span*
by (*metis* (*no-types*, *lifting*) *imageE*)
have *1* = *a* $ *least-n* **using** *rref-condition2*[*OF rref-A*] *row-i-A-not-zero* **unfolding** *least-n-def a-eq-Ai* **by** *presburger*
also have... = (∑ *v*∈(*R* − {*a*}). *f v* ∗*s v*) $ *least-n* **using** *f* **by** *auto*
also have ... = (∑ *v*∈(*R* − {*a*}). (*f v* ∗*s v*) $ *least-n*) **unfolding** *sum-component*
..
also have ... = (∑ *v*∈(*R* − {*a*}). (*f v*) ∗ (*v* $ *least-n*)) **unfolding** *vector-smult-component*

```
..
  also have ... = (∑ v∈(R − {a}). 0)
  proof (rule sum.cong)
    fix x assume x: x ∈ R − {a}
    from this obtain j where x-eq-row-j-A: x=row j A unfolding R-def by auto
    hence i-not-j: i ≠ j using a-eq-row-i-A x by auto
     have x-least-is-zero: x $ least-n = 0 using rref-condition4[OF rref-A] i-not-j
row-i-A-not-zero
      unfolding x-eq-row-j-A least-n-def row-def vec-nth-inverse by blast
    show f x * x $ least-n = 0 unfolding x-least-is-zero by auto
  qed rule
  also have ... = 0 unfolding sum.neutral-const ..
  finally show False by simp
qed
```

Here we start to prove that the transformation from the original matrix to its reduced row echelon form has been carried out by means of elementary operations.

The following function eliminates all entries of the j-th column using the non-zero element situated in the position (i,j). It is introduced to make easier the proof that each Gauss-Jordan step consists in applying suitable elementary operations.

```
primrec row-add-iterate :: 'a::{semiring-1, uminus}^'n^'m::{mod-type} => nat
=> 'm => 'n => 'a^'n^'m::{mod-type}
  where row-add-iterate A 0 i j = (if i=0 then A else row-add A 0 i (−A $ 0 $
j))
  | row-add-iterate A (Suc n) i j = (if (Suc n = to-nat i) then row-add-iterate A n
i j
     else row-add-iterate (row-add A (from-nat (Suc n)) i (− A $ (from-nat (Suc
n)) $ j)) n i j)

lemma invertible-row-add-iterate:
  fixes A::'a::{ring-1}^'n^'m::{mod-type}
  assumes n: n<nrows A
  shows ∃ P. invertible P ∧ row-add-iterate A n i j = P**A
  using n
proof (induct n arbitrary: A)
  fix A::'a::{ring-1}^'n^'m::{mod-type}
  show ∃ P. invertible P ∧ row-add-iterate A 0 i j = P ** A
  proof (cases i=0)
    case True show ?thesis
     unfolding row-add-iterate.simps by (metis True invertible-def matrix-mul-lid)
  next
    case False
   show ?thesis by (metis False invertible-row-add row-add-iterate.simps(1) row-add-mat-1)

  qed
  fix n and A::'a::{ring-1}^'n^'m::{mod-type}
```

define A' **where** $A' = \mathit{row\text{-}add}\ A\ (\mathit{from\text{-}nat}\ (\mathit{Suc}\ n))\ i\ (-\ A\ \$\ \mathit{from\text{-}nat}\ (\mathit{Suc}\ n)\ \$\ j)$
assume *hyp*: $\bigwedge A{::}'a{::}\{\mathit{ring\text{-}1}\}\hat{}'n\hat{}'m{::}\{\mathit{mod\text{-}type}\}.\ n < \mathit{nrows}\ A \Longrightarrow \exists P.\ \mathit{invertible}\ P \wedge \mathit{row\text{-}add\text{-}iterate}\ A\ n\ i\ j = P\ **\ A$ **and** *Suc-n*: $\mathit{Suc}\ n < \mathit{nrows}\ A$
hence $\exists P.\ \mathit{invertible}\ P \wedge \mathit{row\text{-}add\text{-}iterate}\ A'\ n\ i\ j = P\ **\ A'$ **unfolding** *nrows-def* **by** *auto*
from *this* **obtain** P **where** *inv-P*: *invertible* P **and** P: $\mathit{row\text{-}add\text{-}iterate}\ A'\ n\ i\ j = P\ **\ A'$ **by** *auto*
show $\exists P.\ \mathit{invertible}\ P \wedge \mathit{row\text{-}add\text{-}iterate}\ A\ (\mathit{Suc}\ n)\ i\ j = P\ **\ A$
unfolding *row-add-iterate.simps*
proof (*cases Suc n = to-nat i*)
case *True*
show $\exists P.\ \mathit{invertible}\ P \wedge$
(*if Suc n = to-nat i then row-add-iterate A n i j*
else row-add-iterate (*row-add A* (*from-nat* (*Suc n*)) *i* (− *A* $ *from-nat* (*Suc n*) $ *j*)) *n i j*) =
$P\ **\ A$
unfolding *if-P*[*OF True*] **using** *hyp Suc-n* **by** *simp*
next
case *False*
show $\exists P.\ \mathit{invertible}\ P \wedge$
(*if Suc n = to-nat i then row-add-iterate A n i j*
else row-add-iterate (*row-add A* (*from-nat* (*Suc n*)) *i* (− *A* $ *from-nat* (*Suc n*) $ *j*)) *n i j*) =
$P\ **\ A$
unfolding *if-not-P*[*OF False*]
unfolding P[*unfolded* A'*-def*]
proof (*rule exI*[*of* - $P\ **$ (*row-add* (*mat 1*) (*from-nat* (*Suc n*)) *i* (− *A* $ *from-nat* (*Suc n*) $ *j*))], *rule conjI*)
show *invertible* ($P\ **$ *row-add* (*mat 1*) (*from-nat* (*Suc n*)) *i* (− *A* $ *from-nat* (*Suc n*) $ *j*))
by (*metis False Suc-n inv-P invertible-mult invertible-row-add to-nat-from-nat-id nrows-def*)
show $P\ **$ *row-add A* (*from-nat* (*Suc n*)) *i* (− *A* $ *from-nat* (*Suc n*) $ *j*) =
$P\ **$ *row-add* (*mat 1*) (*from-nat* (*Suc n*)) *i* (− *A* $ *from-nat* (*Suc n*) $ *j*) $**$ *A*
using *matrix-mul-assoc row-add-mat-1*[*of from-nat* (*Suc n*) *i* (− *A* $ *from-nat* (*Suc n*) $ *j*)]
by *metis*
qed
qed
qed

lemma *row-add-iterate-preserves-greater-than-n*:
fixes $A{::}'a{::}\{\mathit{ring\text{-}1}\}\hat{}'n\hat{}'m{::}\{\mathit{mod\text{-}type}\}$
assumes *n*: $n{<}\mathit{nrows}\ A$
and *a*: *to-nat a* $> n$
shows (*row-add-iterate A n i j*) $ *a* $ *b* = *A* $ *a* $ *b*
using *assms*

proof (*induct n arbitrary*: *A*)
case *0*
show *?case* **unfolding** *row-add-iterate.simps*
proof (*auto*)
assume $i \neq 0$
hence $a \neq 0$ **by** (*metis 0.prems*(*2*) *less-numeral-extra*(*3*) *to-nat-0*)
thus *row-add A 0 i* (− *A* $ *0* $ *j*) $ *a* $ *b* = *A* $ *a* $ *b* **unfolding** *row-add-def*
by *auto*
qed
next
fix *n* **and** *A*::*'a*::{*ring-1*}^*n*^*m*::{*mod-type*}
assume *hyp*: (⋀*A*::*'a*::{*ring-1*}^*n*^*m*::{*mod-type*}. *n* < *nrows A* ⟹ *n* < *to-nat a* ⟹ *row-add-iterate A n i j* $ *a* $ *b* = *A* $ *a* $ *b*)
and *suc-n-less-card*: *Suc n* < *nrows A* **and** *suc-n-kess-a*: *Suc n* < *to-nat a*
hence *row-add-iterate-A*: *row-add-iterate A n i j* $ *a* $ *b* = *A* $ *a* $ *b* **by** *auto*
show *row-add-iterate A* (*Suc n*) *i j* $ *a* $ *b* = *A* $ *a* $ *b*
proof (*cases Suc n* = *to-nat i*)
case *True*
show *row-add-iterate A* (*Suc n*) *i j* $ *a* $ *b* = *A* $ *a* $ *b* **unfolding** *row-add-iterate.simps if-P*[*OF True*] **using** *row-add-iterate-A* **.**
next
case *False*
define *A'* **where** *A'* = *row-add A* (*from-nat* (*Suc n*)) *i* (− *A* $ *from-nat* (*Suc n*) $ *j*)
have *row-add-iterate-A'*: *row-add-iterate A' n i j* $ *a* $ *b* = *A'* $ *a* $ *b* **using** *hyp suc-n-less-card suc-n-kess-a* **unfolding** *nrows-def* **by** *auto*
have *from-nat-not-a*: *from-nat* (*Suc n*) ≠ *a* **by** (*metis less-not-refl suc-n-kess-a suc-n-less-card to-nat-from-nat-id nrows-def*)
show *row-add-iterate A* (*Suc n*) *i j* $ *a* $ *b* = *A* $ *a* $ *b* **unfolding** *row-add-iterate.simps if-not-P*[*OF False*] *row-add-iterate-A'*[*unfolded A'-def*]
unfolding *row-add-def* **using** *from-nat-not-a* **by** *simp*
qed
qed

lemma *row-add-iterate-preserves-pivot-row*:
fixes *A*::*'a*::{*ring-1*}^*n*^*m*::{*mod-type*}
assumes *n*: *n*<*nrows A*
and *a*: *to-nat i* ≤ *n*
shows (*row-add-iterate A n i j*) $ *i* $ *b* = *A* $ *i* $ *b*
using *assms*
proof (*induct n arbitrary*: *A*)
case *0*
show *?case* **by** (*metis 0.prems*(*2*) *le-0-eq least-mod-type row-add-iterate.simps*(*1*) *to-nat-eq to-nat-mono'*)
next
fix *n* **and** *A*::*'a*::{*ring-1*}^*n*^*m*::{*mod-type*}
assume *hyp*: ⋀*A*::*'a*::{*ring-1*}^*n*^*m*::{*mod-type*}. *n* < *nrows A* ⟹ *to-nat i* ≤ *n* ⟹ *row-add-iterate A n i j* $ *i* $ *b* = *A* $ *i* $ *b*

and *Suc-n-less-card*: *Suc n* < *nrows A* **and** *i-less-suc*: *to-nat i* ≤ *Suc n*
show *row-add-iterate A* (*Suc n*) *i j* $ *i* $ *b* = *A* $ *i* $ *b*
proof (*cases Suc n* = *to-nat i*)
case *True*
show *?thesis* **unfolding** *row-add-iterate.simps if-P*[*OF True*] **apply** (*rule row-add-iterate-preserves-greater-than-n*) **using** *Suc-n-less-card True lessI* **by** *linarith+*
next
case *False*
define *A'* **where** *A'* = *row-add A* (*from-nat* (*Suc n*)) *i* (− *A* $ *from-nat* (*Suc n*) $ *j*)
have *row-add-iterate-A'*: *row-add-iterate A' n i j* $ *i* $ *b* = *A'* $ *i* $ *b* **using** *hyp Suc-n-less-card i-less-suc False* **unfolding** *nrows-def* **by** *auto*
have *from-nat-noteq-i*: *from-nat* (*Suc n*) ≠ *i* **using** *False Suc-n-less-card from-nat-not-eq* **unfolding** *nrows-def* **by** *blast*
show *?thesis* **unfolding** *row-add-iterate.simps if-not-P*[*OF False*] *row-add-iterate-A'*[*unfolded A'-def*]
unfolding *row-add-def* **using** *from-nat-noteq-i* **by** *simp*
qed
qed

lemma *row-add-iterate-eq-row-add*:
fixes *A*::*'a*::{*ring-1*}^*n*^*m*::{*mod-type*}
assumes *a-not-i*: *a* ≠ *i*
and *n*: *n*<*nrows A*
and *to-nat a* ≤ *n*
shows (*row-add-iterate A n i j*) $ *a* $ *b* = (*row-add A a i* (− *A* $ *a* $ *j*)) $ *a* $ *b*
using *assms*
proof (*induct n arbitrary*: *A*)
case *0*
show *?case* **unfolding** *row-add-iterate.simps* **using** *0.prems*(*3*) *a-not-i to-nat-eq-0 least-mod-type* **by** *force*
next
fix *n* **and** *A*::*'a*::{*ring-1*}^*n*^*m*::{*mod-type*}
assume *hyp*: (⋀*A*::*'a*::{*ring-1*}^*n*^*m*::{*mod-type*}. *a* ≠ *i* ⟹ *n* < *nrows A* ⟹ *to-nat a* ≤ *n*
⟹ *row-add-iterate A n i j* $ *a* $ *b* = *row-add A a i* (− *A* $ *a* $ *j*) $ *a* $ *b*)
and *a-not-i*: *a* ≠ *i*
and *suc-n-less-card*: *Suc n* < *nrows A*
and *a-le-suc-n*: *to-nat a* ≤ *Suc n*
show *row-add-iterate A* (*Suc n*) *i j* $ *a* $ *b* = *row-add A a i* (− *A* $ *a* $ *j*) $ *a* $ *b*
proof (*cases Suc n* = *to-nat i*)
case *True*
show *row-add-iterate A* (*Suc n*) *i j* $ *a* $ *b* = *row-add A a i* (− *A* $ *a* $ *j*) $ *a* $ *b* **unfolding** *row-add-iterate.simps if-P*[*OF True*]
apply (*rule hyp*[*OF a-not-i*], *auto simp add*: *Suc-lessD suc-n-less-card*) **by** (*metis True a-le-suc-n a-not-i le-SucE to-nat-eq*)
next

case *False* **note** *Suc-n-not-i=False*
show *?thesis* **unfolding** *row-add-iterate.simps if-not-P[OF False]*
proof (*cases to-nat a = Suc n*) **case** *True*
show *row-add-iterate* (*row-add A* (*from-nat* (*Suc n*)) *i* (− *A* $ *from-nat* (*Suc n*) $ *j*)) *n i j* $ *a* $ *b* = *row-add A a i* (− *A* $ *a* $ *j*) $ *a* $ *b*
by (*metis Suc-le-lessD True order-refl less-imp-le row-add-iterate-preserves-greater-than-n suc-n-less-card to-nat-from-nat nrows-def*)
next
case *False*
define *A′* **where** *A′* = *row-add A* (*from-nat* (*Suc n*)) *i* (− *A* $ *from-nat* (*Suc n*) $ *j*)
have *rw*: *row-add-iterate A′ n i j* $ *a* $ *b* = *row-add A′ a i* (− *A′* $ *a* $ *j*) $ *a* $ *b*
proof (*rule hyp*)
show *a* ≠ *i* **using** *a-not-i* **.**
show *n* < *nrows A′* **using** *suc-n-less-card* **unfolding** *nrows-def* **by** *auto*
show *to-nat a* ≤ *n* **using** *False a-le-suc-n* **by** *simp*
qed
have *rw1*: *row-add A* (*from-nat* (*Suc n*)) *i* (− *A* $ *from-nat* (*Suc n*) $ *j*) $ *a* $ *b* = *A* $ *a* $ *b*
unfolding *row-add-def* **using** *False suc-n-less-card* **unfolding** *nrows-def* **by** (*auto simp add*: *to-nat-from-nat-id*)
have *rw2*: *row-add A* (*from-nat* (*Suc n*)) *i* (− *A* $ *from-nat* (*Suc n*) $ *j*) $ *a* $ *j* = *A* $ *a* $ *j*
unfolding *row-add-def* **using** *False suc-n-less-card* **unfolding** *nrows-def* **by** (*auto simp add*: *to-nat-from-nat-id*)
have *rw3*: *row-add A* (*from-nat* (*Suc n*)) *i* (− *A* $ *from-nat* (*Suc n*) $ *j*) $ *i* $ *b* = *A* $ *i* $ *b*
unfolding *row-add-def* **using** *Suc-n-not-i suc-n-less-card* **unfolding** *nrows-def* **by** (*auto simp add*: *to-nat-from-nat-id*)
show *row-add-iterate A′ n i j* $ *a* $ *b* = *row-add A a i* (− *A* $ *a* $ *j*) $ *a* $ *b*
unfolding *rw row-add-def* **apply** *simp*
unfolding *A′-def rw1 rw2 rw3* **..**
qed
qed
qed

lemma *row-add-iterate-eq-Gauss-Jordan-in-ij*:
fixes *A*::*′a*::{*field*}^*′n*^*′m*::{*mod-type*} **and** *i*::*′m* **and** *j*::*′n*
defines *A′*: *A′*== *mult-row* (*interchange-rows A i* (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*)) *i* (*1* / (*interchange-rows A i* (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*)) $ *i* $ *j*)
shows *row-add-iterate A′* (*nrows A* − *1*) *i j* = *Gauss-Jordan-in-ij A i j*
proof (*unfold Gauss-Jordan-in-ij-def Let-def*, *vector*, *auto*)
fix *ia*
have *interchange-rw*: *A* $ (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) $ *j* = *interchange-rows A i* (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) $ *i* $ *j*
using *interchange-rows-j*[*symmetric*, *of A* (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*)]
by *auto*

show *row-add-iterate A′ (nrows A − Suc 0) i j $ i $ ia =*
mult-row (interchange-rows A i (LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n)) i (1 / A $ (LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n) $ j) $ i $ ia
unfolding *interchange-rw* **unfolding** *A′*
proof (*rule row-add-iterate-preserves-pivot-row*, *unfold nrows-def*)
show *CARD('m) − Suc 0 < CARD('m)* **by** *simp*
have *to-nat i < CARD('m)* **using** *bij-to-nat*[**where** *?'a='m*] **unfolding** *bij-betw-def* **by** *auto*
thus *to-nat i ≤ CARD('m) − Suc 0* **by** *auto*
qed
next
fix *ia iaa*
have *interchange-rw*: *A $ (LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n) $ j = interchange-rows A i (LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n) $ i $ j*
using *interchange-rows-j*[*symmetric*, *of A (LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n)*] **by** *auto*
assume *ia-not-i*: *ia ≠ i*
have *rw*: (− *interchange-rows A i (LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n) $ ia $ j)*
= − mult-row (interchange-rows A i (LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n)) i (1 / interchange-rows A i (LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n) $ i $ j) $ ia $ j
unfolding *interchange-rows-def mult-row-def* **using** *ia-not-i* **by** *auto*
show *row-add-iterate A′ (nrows A − Suc 0) i j $ ia $ iaa =*
row-add (mult-row (interchange-rows A i (LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n)) i (1 / A $ (LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n) $ j)) ia i
(− interchange-rows A i (LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n) $ ia $ j) $
ia $
iaa
unfolding *interchange-rw A′ rw*
proof (*rule row-add-iterate-eq-row-add*[*of ia i (nrows A − Suc 0) - j iaa*], *unfold nrows-def*)
show *ia ≠ i* **using** *ia-not-i* **.**
show *CARD('m) − Suc 0 < CARD('m)* **by** *simp*
have *to-nat ia < CARD('m)* **using** *bij-to-nat*[**where** *?'a='m*] **unfolding** *bij-betw-def* **by** *auto*
thus *to-nat ia ≤ CARD('m) − Suc 0* **by** *simp*
qed
qed

lemma *invertible-Gauss-Jordan-column-k*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'m*::{*mod-type*} **and** *k*::*nat*
shows ∃ *P*. *invertible P ∧ (snd (Gauss-Jordan-column-k (i,A) k)) = P**A*
unfolding *Gauss-Jordan-column-k-def Let-def*
proof (*auto*)
show ∃ *P*. *invertible P ∧ A = P ** *A* **and** ∃ *P*. *invertible P ∧ A = P ** *A*
using *invertible-mat-1 matrix-mul-lid*[*of A*] **by** *auto*
next
fix *m*

assume *i*: $i \neq nrows\ A$
and *i-le-m*: $from\text{-}nat\ i \leq m$ **and** *Amk-not-zero*: $A\ \$\ m\ \$\ from\text{-}nat\ k \neq 0$
define *A-interchange* **where** $A\text{-}interchange = interchange\text{-}rows\ A\ (from\text{-}nat\ i)\ (LEAST\ n.\ A\ \$\ n\ \$\ from\text{-}nat\ k \neq 0 \wedge (from\text{-}nat\ i) \leq n)$
define *A-mult* **where** $A\text{-}mult = mult\text{-}row\ A\text{-}interchange\ (from\text{-}nat\ i)\ (1\ /\ (A\text{-}interchange\ \$\ (from\text{-}nat\ i)\ \$\ from\text{-}nat\ k))$
obtain *P* **where** *inv-P*: *invertible P* **and** *PA*: $A\text{-}interchange = P{**}A$
unfolding *A-interchange-def*
using *interchange-rows-mat-1*[*of from-nat i* (*LEAST n. A* $ *n* $ *from-nat k* $\neq$ *0* $\wedge$ *from-nat i* $\leq$ *n*) *A*]
using *invertible-interchange-rows*[*of from-nat i* (*LEAST n. A* $ *n* $ *from-nat k* $\neq$ *0* $\wedge$ *from-nat i* $\leq$ *n*)]
by *fastforce*
define $Q :: 'a\hat{}'m\hat{}'m$
where $Q = mult\text{-}row\ (mat\ 1)\ (from\text{-}nat\ i)\ (1\ /\ (A\text{-}interchange\ \$\ (from\text{-}nat\ i)\ \$\ from\text{-}nat\ k))$
have *Q-A-interchange*: $A\text{-}mult = Q{**}A\text{-}interchange$ **unfolding** *A-mult-def A-interchange-def Q-def* **unfolding** *mult-row-mat-1* **..**
have *inv-Q*: *invertible Q*
proof (*unfold Q-def*, *rule invertible-mult-row'*, *unfold A-interchange-def*, *rule LeastI2-ex*)
show $\exists a.\ A\ \$\ a\ \$\ from\text{-}nat\ k \neq 0 \wedge (from\text{-}nat\ i) \leq a$ **using** *i-le-m Amk-not-zero* **by** *blast*
show $\bigwedge x.\ A\ \$\ x\ \$\ from\text{-}nat\ k \neq 0 \wedge (from\text{-}nat\ i) \leq x \Longrightarrow 1\ /\ interchange\text{-}rows\ A\ (from\text{-}nat\ i)\ x\ \$\ (from\text{-}nat\ i)\ \$\ from\text{-}nat\ k \neq 0$
using *interchange-rows-i mult-zero-left nonzero-divide-eq-eq zero-neq-one* **by** *fastforce*
qed
obtain *Pa* **where** *inv-Pa*: *invertible Pa* **and** *Pa*: $row\text{-}add\text{-}iterate\ (Q\ **\ (P\ **\ A))\ (nrows\ A - 1)\ (from\text{-}nat\ i)\ (from\text{-}nat\ k) = Pa\ **\ (Q\ **\ (P\ **\ A))$
using *invertible-row-add-iterate* **by** (*metis* (*full-types*) *diff-less nrows-def zero-less-card-finite zero-less-one*)
show $\exists P.\ invertible\ P \wedge Gauss\text{-}Jordan\text{-}in\text{-}ij\ A\ (from\text{-}nat\ i)\ (from\text{-}nat\ k) = P\ **\ A$
proof (*rule exI*[*of - Pa***Q***P*], *rule conjI*)
show *invertible* (*Pa* ** *Q* ** *P*) **using** *inv-P inv-Pa inv-Q invertible-mult* **by** *auto*
have $Gauss\text{-}Jordan\text{-}in\text{-}ij\ A\ (from\text{-}nat\ i)\ (from\text{-}nat\ k) = row\text{-}add\text{-}iterate\ A\text{-}mult\ (nrows\ A - 1)\ (from\text{-}nat\ i)\ (from\text{-}nat\ k)$
unfolding *row-add-iterate-eq-Gauss-Jordan-in-ij*[*symmetric*] *A-mult-def A-interchange-def* **..**
also have ... = *Pa* ** (*Q* ** (*P* ** *A*)) **using** *Pa* **unfolding** *PA*[*symmetric*] *Q-A-interchange*[*symmetric*] **.**
also have ... = *Pa* ** *Q* ** *P* ** *A* **unfolding** *matrix-mul-assoc* **..**
finally show $Gauss\text{-}Jordan\text{-}in\text{-}ij\ A\ (from\text{-}nat\ i)\ (from\text{-}nat\ k) = Pa\ **\ Q\ **\ P\ **\ A$ **.**
qed
qed

lemma *invertible-Gauss-Jordan-up-to-k*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'m*::{*mod-type*}
shows ∃ *P*. *invertible P* ∧ (*Gauss-Jordan-upt-k A k*) = *P*∗∗*A*
proof (*induct k*)
case *0*
have *rw*: [*0*..<*Suc 0*] = [*0*] **by** *fastforce*
show *?case*
unfolding *Gauss-Jordan-upt-k-def rw foldl.simps*
using *invertible-Gauss-Jordan-column-k* **.**
case (*Suc k*)
have *rw2*: [*0*..<*Suc* (*Suc k*)] = [*0*..< *Suc k*] @ [(*Suc k*)] **by** *simp*
obtain *P'* **where** *inv-P'*: *invertible P'* **and** *Gk-eq-P'A*: *Gauss-Jordan-upt-k A k* = *P'* ∗∗ *A* **using** *Suc.hyps* **by** *force*
have *g*: *Gauss-Jordan-upt-k A k* = *snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc k*]) **unfolding** *Gauss-Jordan-upt-k-def* **by** *auto*
show *?case* **unfolding** *Gauss-Jordan-upt-k-def* **unfolding** *rw2 foldl-append foldl.simps*
apply (*subst prod.collapse*[*symmetric*, *of* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc k*]), *unfolded g*[*symmetric*]])
using *invertible-Gauss-Jordan-column-k*
using *Suc.hyps* **using** *invertible-mult matrix-mul-assoc* **by** *metis*
qed

lemma *inj-index-independent-rows*:
fixes *A*::*'a*::{*field*}^*'m*::{*mod-type*}^*'n*::{*finite*,*one*,*plus*,*ord*}
assumes *rref-A*: *reduced-row-echelon-form A*
and *x*: *row x A* ∈ {*row i A* |*i*. *row i A* ≠ *0*}
and *eq*: *A* $ *x* = *A* $ *y*
shows *x* = *y*
proof (*rule ccontr*)
assume *x-not-y*: *x* ≠ *y*
have *not-zero-x*: ¬ *is-zero-row x A*
using *x* **unfolding** *is-zero-row-def* **unfolding** *is-zero-row-upt-k-def* **unfolding** *row-def vec-eq-iff*
ncols-def
by *auto*
hence *not-zero-y*: ¬ *is-zero-row y A* **using** *eq* **unfolding** *is-zero-row-def'* **by** *simp*
have *Ax*: *A* $ *x* $ (*LEAST k*. *A* $ *x* $ *k* ≠ *0*) = *1* **using** *not-zero-x rref-condition2*[*OF rref-A*] **by** *simp*
have *Ay*: *A* $ *x* $ (*LEAST k*. *A* $ *y* $ *k* ≠ *0*) = *0* **using** *not-zero-y x-not-y rref-condition4*[*OF rref-A*] **by** *fast*
show *False* **using** *Ax Ay* **unfolding** *eq* **by** *simp*
qed

The final results:

lemma *invertible-Gauss-Jordan*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'m*::{*mod-type*}

shows $\exists P.$ *invertible* $P \wedge$ (*Gauss-Jordan* A) $= P{*}{*}A$ **unfolding** *Gauss-Jordan-def*
using *invertible-Gauss-Jordan-up-to-k* **.**

lemma *Gauss-Jordan*:
fixes A::$'a$::{*field*}$^{\frown\prime}n$::{*mod-type*}$^{\frown\prime}m$::{*mod-type*}
shows $\exists P.$ *invertible* $P \wedge$ (*Gauss-Jordan* A) $= P{*}{*}A \wedge$ *reduced-row-echelon-form*
(*Gauss-Jordan* A)
by (*simp add*: *invertible-Gauss-Jordan rref-Gauss-Jordan*)

Some properties about the rank of a matrix, obtained thanks to the Gauss-Jordan algorithm and the reduced row echelon form.

lemma *rref-rank*:
fixes A::$'a$::{*field*}$^{\frown\prime}m$::{*mod-type*}$^{\frown\prime}n$::{*finite*,*one*,*plus*,*ord*}
assumes *rref-A*: *reduced-row-echelon-form* A
shows *rank* A = *card* {*row* i A |i. *row* i $A \neq 0$}
unfolding *rank-def row-rank-def*
proof (*rule vec.dim-unique*[*of* {*row* i A | i. *row* i $A \neq 0$}])
show {*row* i A |i. *row* i $A \neq 0$} ⊆ *row-space* A
proof (*auto*, *unfold row-space-def rows-def*)
fix i **assume** *row* i $A \neq 0$ **show** *row* i $A \in$ *vec.span* {*row* i A |i. $i \in$ *UNIV*}
by (*rule vec.span-base*, *auto*)
qed
show *row-space* $A \subseteq$ *vec.span* {*row* i A |i. *row* i $A \neq 0$}
proof (*unfold row-space-def rows-def*, *cases* $\exists i.$ *row* i $A = 0$)
case *True*
have *set-rw*: {*row* i A |i. $i \in$ *UNIV*} = *insert* 0 {*row* i A |i. *row* i $A \neq 0$}
using *True* **by** *auto*
have *vec.span* {*row* i A |i. $i \in$ *UNIV*} = *vec.span* {*row* i A |i. *row* i $A \neq 0$}
unfolding *set-rw* **using** *vec.span-insert-0* **.**
thus *vec.span* {*row* i A |i. $i \in$ *UNIV*} ⊆ *vec.span* {*row* i A |i. *row* i $A \neq 0$}
by *simp*
next
case *False* **show** *vec.span* {*row* i A |i. $i \in$ *UNIV*} ⊆ *vec.span* {*row* i A |i. *row*
i $A \neq 0$} **using** *False* **by** *simp*
qed
show *vec.independent* {*row* i A |i. *row* i $A \neq 0$} **by** (*rule independent-not-zero-rows-rref*[*OF*
rref-A])
show *card* {*row* i A |i. *row* i $A \neq 0$} = *card* {*row* i A |i. *row* i $A \neq 0$} **..**
qed

lemma *column-leading-coefficient-component-eq*:
fixes A::$'a$::{*field*}$^{\frown\prime}m$::{*mod-type*}$^{\frown\prime}n$::{*finite*,*one*,*plus*,*ord*}
assumes *rref-A*: *reduced-row-echelon-form* A
and v: $v \in$ {*column* (*LEAST* n. A \$ i \$ $n \neq 0$) A |i. *row* i $A \neq 0$}
and *vx*: v \$ $x \neq 0$
and *vy*: v \$ $y \neq 0$
shows $x = y$
proof −
obtain b **where** b: v = *column* (*LEAST* n. A \$ b \$ $n \neq 0$) A **and** *row-b*: *row* b

$A \neq 0$ **using** v **by** *blast*
have *vb-not-zero*: $v \$ b \neq 0$ **unfolding** b *column-def* **by** (*auto*, *metis is-zero-row-eq-row-zero row-b rref-A rref-condition2 zero-neq-one*)
have *b-eq-x*: $b = x$
proof (*rule ccontr*)
assume *b-not-x*: $b \neq x$
have $A \$ x \$ (LEAST\ n.\ A \$ b \$ n \neq 0) = 0$
by (*rule rref-condition4-explicit*[*OF rref-A - b-not-x*], *simp add*: *is-zero-row-eq-row-zero row-b*)
thus *False* **using** *vx* **unfolding** *b column-def* **by** *auto*
qed
moreover have *b-eq-y*: $b = y$
proof (*rule ccontr*)
assume *b-not-y*: $b \neq y$
have $A \$ y \$ (LEAST\ n.\ A \$ b \$ n \neq 0) = 0$
by (*rule rref-condition4-explicit*[*OF rref-A - b-not-y*], *simp add*: *is-zero-row-eq-row-zero row-b*)
thus *False* **using** *vy* **unfolding** *b column-def* **by** *auto*
qed
ultimately show *?thesis* **by** *simp*
qed

lemma *column-leading-coefficient-component-1*:
fixes $A::'a::\{field\}\hat{}'m::\{mod\text{-}type\}\hat{}'n::\{finite,one,plus,ord\}$
assumes *rref-A*: *reduced-row-echelon-form A*
and *v*: $v \in \{column\ (LEAST\ n.\ A \$ i \$ n \neq 0)\ A \mid i.\ row\ i\ A \neq 0\}$
and *vx*: $v \$ x \neq 0$
shows $v \$ x = 1$
proof −
obtain b **where** *b*: $v = column\ (LEAST\ n.\ A \$ b \$ n \neq 0)\ A$ **and** *row-b*: *row b A* $\neq 0$ **using** *v* **by** *blast*
have *vb-not-zero*: $v \$ b \neq 0$ **unfolding** b *column-def* **by** (*auto*, *metis is-zero-row-eq-row-zero row-b rref-A rref-condition2 zero-neq-one*)
have *b-eq-x*: $b = x$
by (*metis b column-def is-zero-row-eq-row-zero row-b rref-A rref-condition4 transpose-row-code transpose-row-def vx*)
show *?thesis*
using *rref-condition2-explicit*[*OF rref-A, of b*] *row-b*
unfolding *b column-def is-zero-row-def'*
by (*metis* (*mono-tags*) ‹¬ *is-zero-row b A* ⟹ $A \$ b \$ (LEAST\ k.\ A \$ b \$ k \neq 0) = 1$›
b-eq-x is-zero-row-eq-row-zero vec-lambda-beta)
qed

lemma *column-leading-coefficient-component-0*:
fixes $A::'a::\{field\}\hat{}'m::\{mod\text{-}type\}\hat{}'n::\{finite,one,plus,ord\}$
assumes *rref-A*: *reduced-row-echelon-form A*

and *v*: $v \in \{column\ (LEAST\ n.\ A\ \$\ i\ \$\ n \neq 0)\ A\ |i.\ row\ i\ A \neq 0\}$
and *vx*: $v\ \$\ x \neq 0$
and *x-not-y*: $x \neq y$
shows $v\ \$\ y = 0$ **using** *column-leading-coefficient-component-eq*[*OF rref-A v vx*] *x-not-y* **by** *auto*

lemma *rref-col-rank*:
fixes *A*::'*a*::{*field*}^'*m*::{*mod-type*}^'*n*::{*mod-type*}
assumes *rref-A*: *reduced-row-echelon-form A*
shows *col-rank A* = *card* {*column* (*LEAST n. A* $ *i* $ *n* ≠ *0*) *A* | *i. row i A* ≠ *0*}
proof (*unfold col-rank-def*, *rule vec.dim-unique*[*of* {*column* (*LEAST n. A* $ *i* $ *n* ≠ *0*) *A* | *i. row i A* ≠ *0*}])
show {*column* (*LEAST n. A* $ *i* $ *n* ≠ *0*) *A* |*i. row i A* ≠ *0*} ⊆ *col-space A*
by (*auto simp add*: *col-space-def*, *rule vec.span-base*, *unfold columns-def*, *auto*)
show *vec.independent* {*column* (*LEAST n. A* $ *i* $ *n* ≠ *0*) *A* |*i. row i A* ≠ *0*}
proof (*rule vec.independent-if-scalars-zero*, *auto*)
fix *f i*
let *?x* = *column* (*LEAST n. A* $ *i* $ *n* ≠ *0*) *A*
have *sum0*: (∑ *x*∈{*column* (*LEAST n. A* $ *i* $ *n* ≠ *0*) *A* |*i. row i A* ≠ *0*} − {*?x*}. *f x* ∗ (*x* $ *i*)) = *0*
proof (*rule sum.neutral*, *rule ballI*)
fix *x* **assume** *x*: *x* ∈ {*column* (*LEAST n. A* $ *i* $ *n* ≠ *0*) *A* |*i. row i A* ≠ *0*} − {*?x*}
obtain *j* **where** *x-eq*: *x*=*column* (*LEAST n. A* $ *j* $ *n* ≠ *0*) *A* **and** *row-j-not-0*: *row j A* ≠ *0*
and *j-not-i*: *j*≠*i* **using** *x* **by** *auto*
have *x*$*i*=*0* **unfolding** *x-eq column-def*
by (*auto*, *metis is-zero-row-eq-row-zero j-not-i row-j-not-0 rref-A rref-condition4-explicit*)
thus *f x* ∗ *x* $ *i* = *0* **by** *simp*
qed
assume *eq-0*: (∑ *x*∈{*column* (*LEAST n. A* $ *i* $ *n* ≠ *0*) *A* |*i. row i A* ≠ *0*}. *f x* ∗*s x*) = *0*
and *i*: *row i A* ≠ *0*
have *xi-1*: (*?x* $ *i*) = *1* **unfolding** *column-def* **by** (*auto*, *metis i is-zero-row-eq-row-zero rref-A rref-condition2-explicit*)
have *0* = (∑ *x*∈{*column* (*LEAST n. A* $ *i* $ *n* ≠ *0*) *A* |*i. row i A* ≠ *0*}. *f x* ∗*s x*) $ *i*
using *eq-0* **by** *auto*
also have ... = (∑ *x*∈{*column* (*LEAST n. A* $ *i* $ *n* ≠ *0*) *A* |*i. row i A* ≠ *0*}. *f x* ∗ (*x* $ *i*))
unfolding *sum-component vector-smult-component* **..**
also have ... = *f ?x* ∗ (*?x* $ *i*)
+ (∑ *x*∈{*column* (*LEAST n. A* $ *i* $ *n* ≠ *0*) *A* |*i. row i A* ≠ *0*} − {*?x*}. *f x* ∗ (*x* $ *i*))
by (*rule sum.remove*, *auto*, *rule exI*[*of* - *i*], *simp add*: *i*)
also have ... = *f ?x* ∗ (*?x* $ *i*) **unfolding** *sum0* **by** *simp*
also have ... = *f* (*column* (*LEAST n. A* $ *i* $ *n* ≠ *0*) *A*) **unfolding** *xi-1* **by** *simp*

```
    finally show f (column (LEAST n. A $ i $ n ≠ 0) A) = 0 by simp
    qed
  show col-space A ⊆ vec.span {column (LEAST n. A $ i $ n ≠ 0) A |i. row i A
≠ 0}
  unfolding col-space-def
  proof (rule vec.span-mono[of (columns A)
       vec.span {column (LEAST n. A $ i $ n ≠ 0) A |i. row i A ≠ 0}, unfolded
vec.span-span], auto)
  fix x assume x: x ∈ columns A
  have f: finite {column (LEAST n. A $ i $ n ≠ 0) A |i. row i A ≠ 0} by simp
  let ?f=λv. x $ (THE i. v $ i ≠ 0)
  show x ∈ vec.span {column (LEAST n. A $ i $ n ≠ 0) A |i. row i A ≠ 0}
   unfolding vec.span-finite[OF f] image-iff bex-UNIV
  proof (rule exI[of - ?f], subst (1) vec-eq-iff, clarify)
   fix i
   show x $ i = (∑ v∈{column (LEAST n. A $ i $ n ≠ 0) A |i. row i A ≠ 0}.
x $ (THE i. v $ i ≠ 0) *s v) $ i
   proof (cases ∃ v. v ∈ {column (LEAST n. A $ i $ n ≠ 0) A |i. row i A ≠ 0}
∧ v $ i ≠ 0)
    case False
    have xi-0: x $ i = 0
    proof (rule ccontr)
     assume xi-not-0: x $ i ≠ 0
       hence row-iA-not-zero: row i A ≠ 0 using x unfolding columns-def
column-def row-def by (vector, metis vec-lambda-unique)
     let ?v=column (LEAST n. A $ i $ n ≠ 0) A
     have ?v ∈ {column (LEAST n. A $ i $ n ≠ 0) A |i. row i A ≠ 0} using
row-iA-not-zero by auto
      moreover have ?v $ i = 1 unfolding column-def by (auto, metis
is-zero-row-eq-row-zero row-iA-not-zero rref-A rref-condition2)
     ultimately show False using False by auto
    qed
    show ?thesis
     unfolding xi-0
   proof (unfold sum-component vector-smult-component, rule sum.neutral[symmetric],
rule ballI)
     fix xa assume xa: xa ∈ {column (LEAST n. A $ i $ n ≠ 0) A |i. row i A
≠ 0}
     have xa $ i = 0 using False xa by auto
     thus x $ (THE i. xa $ i ≠ 0) * xa $ i = 0 by simp
    qed
   next
    case True
    obtain v where v: v ∈ {column (LEAST n. A $ i $ n ≠ 0) A |i. row i A ≠
0} and vi: v $ i ≠ 0
     using True by blast
    obtain b where b: v = column (LEAST n. A $ b $ n ≠ 0) A and row-b:
row b A ≠ 0 using v by blast
   have vb: v $ b ≠ 0 unfolding b column-def by (auto, metis is-zero-row-eq-row-zero
```

row-b rref-A rref-condition2 zero-neq-one)
have *b-eq-i*: $b = i$ **by** (*rule column-leading-coefficient-component-eq*[*OF rref-A v vb vi*])
have *the-vi*: $(THE\ a.\ v\ \$\ a \neq 0) = i$
proof (*rule the-equality*, *rule vi*)
fix *a* **assume** *va*: $v\ \$\ a \neq 0$ **show** *a=i* **by** (*rule column-leading-coefficient-component-eq*[*OF rref-A v va vi*])
qed
have *vi-1*: $v\ \$\ i = 1$ **by** (*rule column-leading-coefficient-component-1*[*OF rref-A v vi*])
have *sum0*: $(\sum v \in \{column\ (LEAST\ n.\ A\ \$\ i\ \$\ n \neq 0)\ A\ |i.\ row\ i\ A \neq 0\} - \{v\}.\ x\ \$\ (THE\ a.\ v\ \$\ a \neq 0) * (v\ \$\ i)) = 0$
proof (*rule sum.neutral*, *rule ballI*)
fix *xa* **assume** *xa*: $xa \in \{column\ (LEAST\ n.\ A\ \$\ i\ \$\ n \neq 0)\ A\ |i.\ row\ i\ A \neq 0\} - \{v\}$
obtain *y* **where** *y*: $xa = column\ (LEAST\ n.\ A\ \$\ y\ \$\ n \neq 0)\ A$ **and** *row-b*: $row\ y\ A \neq 0$ **using** *xa* **by** *blast*
have *xa-in-V*: $xa \in \{column\ (LEAST\ n.\ A\ \$\ i\ \$\ n \neq 0)\ A\ |i.\ row\ i\ A \neq 0\}$ **using** *xa* **by** *simp*
have $xa\ \$\ i = 0$
proof (*rule column-leading-coefficient-component-0*[*OF rref-A xa-in-V*])
show $xa\ \$\ y \neq 0$ **unfolding** *y column-def*
by (*auto*, *metis* (*lifting*, *full-types*) *LeastI2-ex is-zero-row-def' is-zero-row-eq-row-zero row-b*)
have $y \neq b$ **by** (*metis* (*mono-tags*) *Diff-iff b mem-Collect-eq singleton-conv2 xa y*)
thus $y \neq i$ **unfolding** *b-eq-i*[*symmetric*] **.**
qed
thus $x\ \$\ (THE\ a.\ xa\ \$\ a \neq 0) * xa\ \$\ i = 0$ **by** *simp*
qed
have $(\sum v \in \{column\ (LEAST\ n.\ A\ \$\ i\ \$\ n \neq 0)\ A\ |i.\ row\ i\ A \neq 0\}.\ x\ \$\ (THE\ a.\ v\ \$\ a \neq 0) *s v)\ \$\ i =$
$(\sum v \in \{column\ (LEAST\ n.\ A\ \$\ i\ \$\ n \neq 0)\ A\ |i.\ row\ i\ A \neq 0\}.\ x\ \$\ (THE\ a.\ v\ \$\ a \neq 0) * (v\ \$\ i))$
unfolding *sum-component vector-smult-component* **..**
also have $... = x\ \$\ (THE\ a.\ v\ \$\ a \neq 0) * (v\ \$\ i) + (\sum v \in \{column\ (LEAST\ n.\ A\ \$\ i\ \$\ n \neq 0)\ A\ |i.\ row\ i\ A \neq 0\} - \{v\}.\ x\ \$\ (THE\ a.\ v\ \$\ a \neq 0) * (v\ \$\ i))$
by (*simp add*: *sum.remove*[*OF - v*])
also have $... = x\ \$\ (THE\ a.\ v\ \$\ a \neq 0) * (v\ \$\ i)$ **unfolding** *sum0* **by** *simp*
also have $... = x\ \$\ (THE\ a.\ v\ \$\ a \neq 0)$ **unfolding** *vi-1* **by** *simp*
also have $... = x\ \$\ i$ **unfolding** *the-vi* **..**
finally show *?thesis* **by** *simp*
qed
qed
qed
qed (*simp*)

lemma *rref-row-rank*:
fixes *A*::*'a*::{*field*}^*'m*::{*mod-type*}^*'n*::{*finite*,*one*,*plus*,*ord*}
assumes *rref-A*: *reduced-row-echelon-form A*
shows *row-rank A* = *card* {*column* (*LEAST n*. *A* $ *i* $ *n* ≠ *0*) *A* | *i*. *row i A* ≠ *0*}
proof −
let *?f*=λ*x*. *column* ((*LEAST n*. *x* $ *n* ≠ *0*)) *A*
show *?thesis*
unfolding *rref-rank*[*OF rref-A*, *unfolded rank-def*]
proof (*rule bij-betw-same-card*[*of ?f*], *unfold bij-betw-def*, *auto*)
show *inj-on* (λ*x*. *column* (*LEAST n*. *x* $ *n* ≠ *0*) *A*) {*row i A* |*i*. *row i A* ≠ *0*}
unfolding *inj-on-def*
proof (*auto*)
fix *i ia*
assume *i*: *row i A* ≠ *0* **and** *ia*: *row ia A* ≠ *0*
and *c-eq*: *column* (*LEAST n*. *row i A* $ *n* ≠ *0*) *A* = *column* (*LEAST n*. *row ia A* $ *n* ≠ *0*) *A*
show *row i A* = *row ia A*
using *c-eq* **unfolding** *column-def* **unfolding** *row-def vec-nth-inverse*
proof −
have *transpose-row A* (*LEAST R*. *A* $ *ia* $ *R* ≠ *0*) = *transpose-row A* (*LEAST R*. *A* $ *i* $ *R* ≠ *0*)
by (*metis c-eq column-def row-def transpose-row-def vec-nth-inverse*)
hence *f1*: ⋀*x*$_1$. *A* $ *x*$_1$ $ (*LEAST R*. *A* $ *ia* $ *R* ≠ *0*) = *A* $ *x*$_1$ $ (*LEAST R*. *A* $ *i* $ *R* ≠ *0*)
by (*metis* (*no-types*) *transpose-row-def vec-lambda-beta*)
have *f2*: *is-zero-row ia A* = *False*
using *ia is-zero-row-eq-row-zero* **by** *auto*
have *f3*: ¬ *is-zero-row i A*
using *i is-zero-row-eq-row-zero* **by** *auto*
have *A* $ *ia* $ (*LEAST R*. *A* $ *i* $ *R* ≠ *0*) = *1*
using *f1 f2 rref-A rref-condition2* **by** *fastforce*
thus *A* $ *i* = *A* $ *ia*
using *f3 rref-A rref-condition4-explicit* **by** *fastforce*
qed
qed
next
fix *i*
assume *i*: *row i A* ≠ *0*
show ∃*ia*. *column* (*LEAST n*. *row i A* $ *n* ≠ *0*) *A* = *column* (*LEAST n*. *A* $ *ia* $ *n* ≠ *0*) *A* ∧ *row ia A* ≠ *0*
by (*rule exI*[*of* - *i*], *simp add*: *row-def vec-lambda-eta*)
(*metis i is-zero-row-def*′ *is-zero-row-eq-row-zero zero-index*)
next
fix *i*
assume *i*: *row i A* ≠ *0*
show *column* (*LEAST n*. *A* $ *i* $ *n* ≠ *0*) *A* ∈ (λ*x*. *column* (*LEAST n*. *x* $ *n* ≠ *0*) *A*) ‘ {*row i A* |*i*. *row i A* ≠ *0*}
unfolding *column-def row-def image-def*

by (*auto*, *metis i row-def vec-lambda-eta*)
qed
qed

lemma *row-rank-eq-col-rank-rref*:
fixes *A*::*'a*::{*field*}^*'m*::{*mod-type*}^*'n*::{*mod-type*}
assumes *r*: *reduced-row-echelon-form A*
shows *row-rank A* = *col-rank A*
unfolding *rref-row-rank*[*OF r*] *rref-col-rank*[*OF r*] **..**

lemma *row-rank-eq-col-rank*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'m*::{*mod-type*}
shows *row-rank A* = *col-rank A*
proof −
obtain *P* **where** *inv-P*: *invertible P* **and** *G-PA*: (*Gauss-Jordan A*) = *P*∗∗*A*
and *rref-G*: *reduced-row-echelon-form* (*Gauss-Jordan A*)
using *invertible-Gauss-Jordan rref-Gauss-Jordan* **by** *blast*
have *row-rank A* = *row-rank* (*Gauss-Jordan A*)
by (*metis row-space-is-preserved invertible-Gauss-Jordan row-rank-def*)
moreover have *col-rank A* = *col-rank* (*Gauss-Jordan A*)
by (*metis invertible-Gauss-Jordan crk-is-preserved*)
moreover have *col-rank* (*Gauss-Jordan A*) = *row-rank* (*Gauss-Jordan A*)
using *row-rank-eq-col-rank-rref*[*OF rref-G*] **by** *simp*
ultimately show *?thesis* **by** *simp*
qed

theorem *rank-col-rank*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'m*::{*mod-type*}
shows *rank A* = *col-rank A* **unfolding** *rank-def row-rank-eq-col-rank* **..**

theorem *rank-eq-dim-image*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'m*::{*mod-type*}
shows *rank A* = *vec.dim* (*range* (λ*x*. *A* ∗*v x*))
unfolding *rank-col-rank col-rank-def col-space-eq'* **..**

theorem *rank-eq-dim-col-space*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'m*::{*mod-type*}
shows *rank A* = *vec.dim* (*col-space A*) **using** *rank-col-rank* **unfolding** *col-rank-def*
.

lemma *rank-transpose*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'m*::{*mod-type*}
shows *rank* (*transpose A*) = *rank A*
by (*metis rank-def rank-eq-dim-col-space row-rank-def row-space-eq-col-space-transpose*)

lemma *rank-le-nrows*:

fixes $A::'a::\{field\}^{\sim}n::\{mod\text{-}type\}^{\sim}m::\{mod\text{-}type\}$
shows $rank\ A \leq nrows\ A$
unfolding *rank-eq-dim-col-space nrows-def*
by (*metis top-greatest vec.dim-subset vec-dim-card*)

lemma *rank-le-ncols*:
fixes $A::'a::\{field\}^{\sim}n::\{mod\text{-}type\}^{\sim}m::\{mod\text{-}type\}$
shows $rank\ A \leq ncols\ A$
unfolding *rank-def row-rank-def ncols-def*
by (*metis top-greatest vec.dim-subset vec-dim-card*)

lemma *rank-Gauss-Jordan*:
fixes $A::'a::\{field\}^{\sim}n::\{mod\text{-}type\}^{\sim}m::\{mod\text{-}type\}$
shows $rank\ A = rank\ (Gauss\text{-}Jordan\ A)$
by (*metis Gauss-Jordan-def invertible-Gauss-Jordan-up-to-k*
row-rank-eq-col-rank rank-def crk-is-preserved)

Other interesting properties:

lemma *A-0-imp-Gauss-Jordan-0*:
fixes $A::'a::\{field\}^{\sim}n::\{mod\text{-}type\}^{\sim}m::\{mod\text{-}type\}$
assumes $A=0$
shows $Gauss\text{-}Jordan\ A = 0$
proof −
obtain P **where** *PA*: $Gauss\text{-}Jordan\ A = P\ **\ A$ **using** *invertible-Gauss-Jordan*
by *blast*
also have $... = 0$ **unfolding** *assms* **by** (*metis eq-add-iff matrix-add-ldistrib*)
finally show $Gauss\text{-}Jordan\ A = 0$ **.**
qed

lemma *rank-0*: $rank\ 0 = 0$
unfolding *rank-def row-rank-def row-space-def rows-def row-def*
by (*simp add*: *vec.dim-span vec.dim-zero-eq' vec-nth-inverse*)

lemma *rank-greater-zero*:
assumes $A \neq 0$
shows $rank\ A > 0$
proof (*rule ccontr*, *simp*)
assume $rank\ A = 0$
hence $row\text{-}space\ A = \{\} \vee row\text{-}space\ A = \{0\}$ **unfolding** *rank-def row-rank-def*
using *vec.dim-zero-eq* **by** *blast*
hence $row\text{-}space\ A = \{0\}$ **unfolding** *row-space-def* **using** *vec.span-zero* **by** *auto*
hence $rows\ A = \{\} \vee rows\ A = \{0\}$ **unfolding** *row-space-def* **using** *vec.span-0-imp-set-empty-or-0*
by *auto*
hence $rows\ A = \{0\}$ **unfolding** *rows-def row-def* **by** *force*
hence $A = 0$ **unfolding** *rows-def row-def vec-nth-inverse*
by (*auto*, *metis* (*mono-tags*) *mem-Collect-eq singleton-iff vec-lambda-unique*
zero-index)
thus *False* **using** *assms* **by** *contradiction*

qed

lemma *Gauss-Jordan-not-0*:
fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *A* ≠ *0*
shows *Gauss-Jordan A* ≠ *0*
by (*metis assms less-not-refl3 rank-0 rank-Gauss-Jordan rank-greater-zero*)

lemma *rank-eq-suc-to-nat-greatest*:
assumes *A-not-0*: *A* ≠ *0*
shows *rank A* = *to-nat* (*GREATEST a*. ¬ *is-zero-row a* (*Gauss-Jordan A*)) + *1*
proof −
have *rref*: *reduced-row-echelon-form-upt-k* (*Gauss-Jordan A*) (*ncols* (*Gauss-Jordan A*))
using *rref-Gauss-Jordan* **unfolding** *reduced-row-echelon-form-def*
by *auto*
have *not-all-zero*: ¬ (∀ *a*. *is-zero-row-upt-k a* (*ncols* (*Gauss-Jordan A*)) (*Gauss-Jordan A*))
unfolding *is-zero-row-def*[*symmetric*] **using** *Gauss-Jordan-not-0*[*OF A-not-0*] **unfolding** *is-zero-row-def'* **by** (*metis vec-eq-iff zero-index*)
have *rank A* = *card* {*row i* (*Gauss-Jordan A*) |*i*. *row i* (*Gauss-Jordan A*) ≠ *0*}
unfolding *rank-Gauss-Jordan*[*of A*] **unfolding** *rref-rank*[*OF rref-Gauss-Jordan*] **..**
also have ... = *card* {*i*. *i*≤(*GREATEST a*. ¬ *is-zero-row a* (*Gauss-Jordan A*))}
proof (*rule bij-betw-same-card*[*symmetric*, *of* λ*i*. *row i* (*Gauss-Jordan A*)], *unfold bij-betw-def*, *rule conjI*)
show *inj-on* (λ*i*. *row i* (*Gauss-Jordan A*)) {*i*. *i* ≤ (*GREATEST a*. ¬ *is-zero-row a* (*Gauss-Jordan A*))}
proof (*unfold inj-on-def*, *auto*, *rule ccontr*)
fix *x y*
assume *x*: *x* ≤ (*GREATEST a*. ¬ *is-zero-row a* (*Gauss-Jordan A*)) **and** *y*: *y* ≤ (*GREATEST a*. ¬ *is-zero-row a* (*Gauss-Jordan A*))
and *xy-eq-row*: *row x* (*Gauss-Jordan A*) = *row y* (*Gauss-Jordan A*) **and** *x-not-y*: *x* ≠ *y*
show *False*
proof (*cases x*<*y*)
case *True*
have (*LEAST n*. (*Gauss-Jordan A*) $ *x* $ *n* ≠ *0*) < (*LEAST n*. (*Gauss-Jordan A*) $ *y* $ *n* ≠ *0*)
proof (*rule rref-condition3-equiv*[*OF rref-Gauss-Jordan True*])
show ¬ *is-zero-row x* (*Gauss-Jordan A*)
by (*unfold is-zero-row-def*,
rule greatest-ge-nonzero-row'[*OF rref x*[*unfolded is-zero-row-def*] *not-all-zero*])
show ¬ *is-zero-row y* (*Gauss-Jordan A*) **by** (*unfold is-zero-row-def*, *rule greatest-ge-nonzero-row'*[*OF rref y*[*unfolded is-zero-row-def*] *not-all-zero*])
qed
thus *?thesis* **by** (*metis less-irrefl row-def vec-nth-inverse xy-eq-row*)
next
case *False*

hence *x-ge-y*: $x>y$ **using** *x-not-y* **by** *simp*
have (*LEAST n.* (*Gauss-Jordan A*) $ *y* $ *n* ≠ *0*) < (*LEAST n.* (*Gauss-Jordan A*) $ *x* $ *n* ≠ *0*)
proof (*rule rref-condition3-equiv*[*OF rref-Gauss-Jordan x-ge-y*])
show ¬ *is-zero-row x* (*Gauss-Jordan A*)
by (*unfold is-zero-row-def*, *rule greatest-ge-nonzero-row'*[*OF rref x*[*unfolded is-zero-row-def*] *not-all-zero*])
show ¬ *is-zero-row y* (*Gauss-Jordan A*) **by** (*unfold is-zero-row-def*, *rule greatest-ge-nonzero-row'*[*OF rref y*[*unfolded is-zero-row-def*] *not-all-zero*])
qed
thus *?thesis* **by** (*metis less-irrefl row-def vec-nth-inverse xy-eq-row*)
qed
qed
show (λ*i. row i* (*Gauss-Jordan A*)) ‘ {*i. i* ≤ (*GREATEST a.* ¬ *is-zero-row a* (*Gauss-Jordan A*))} = {*row i* (*Gauss-Jordan A*) |*i. row i* (*Gauss-Jordan A*) ≠ *0*}
proof (*unfold image-def*, *auto*)
fix *xa*
assume *xa*: *xa* ≤ (*GREATEST a.* ¬ *is-zero-row a* (*Gauss-Jordan A*))
show ∃*i. row xa* (*Gauss-Jordan A*) = *row i* (*Gauss-Jordan A*) ∧ *row i* (*Gauss-Jordan A*) ≠ *0*
proof (*rule exI*[*of - xa*], *simp*)
have ¬ *is-zero-row xa* (*Gauss-Jordan A*)
by (*unfold is-zero-row-def*, *rule greatest-ge-nonzero-row'*[*OF rref xa*[*unfolded is-zero-row-def*] *not-all-zero*])
thus *row xa* (*Gauss-Jordan A*) ≠ *0* **unfolding** *row-def is-zero-row-def'* **by** (*metis vec-nth-inverse zero-index*)
qed
next
fix *i*
assume *row i* (*Gauss-Jordan A*) ≠ *0*
hence ¬ *is-zero-row i* (*Gauss-Jordan A*) **unfolding** *row-def is-zero-row-def'* **by** (*metis vec-eq-iff vec-nth-inverse zero-index*)
hence *i* ≤ (*GREATEST a.* ¬ *is-zero-row a* (*Gauss-Jordan A*)) **using** *Greatest-ge* **by** *fast*
thus ∃*x*≤*GREATEST a.* ¬ *is-zero-row a* (*Gauss-Jordan A*). *row i* (*Gauss-Jordan A*) = *row x* (*Gauss-Jordan A*)
by *blast*
qed
qed
also have ... = *card* {*i. i* ≤ *to-nat* (*GREATEST a.* ¬ *is-zero-row a* (*Gauss-Jordan A*))}
proof (*rule bij-betw-same-card*[*of* λ*i. to-nat i*], *unfold bij-betw-def*, *rule conjI*)
show *inj-on to-nat* {*i. i* ≤ (*GREATEST a.* ¬ *is-zero-row a* (*Gauss-Jordan A*))} **using** *bij-to-nat* **by** (*metis bij-betw-imp-inj-on subset-inj-on top-greatest*)
show *to-nat* ‘ {*i. i* ≤ (*GREATEST a.* ¬ *is-zero-row a* (*Gauss-Jordan A*))} = {*i. i* ≤ *to-nat* (*GREATEST a.* ¬ *is-zero-row a* (*Gauss-Jordan A*))}
proof (*unfold image-def*, *auto simp add*: *to-nat-mono'*)
fix *x*
assume *x*: *x* ≤ *to-nat* (*GREATEST a.* ¬ *is-zero-row a* (*Gauss-Jordan A*))

hence *from-nat x* ≤ (*GREATEST a.* ¬ *is-zero-row a* (*Gauss-Jordan A*))
by (*metis* (*full-types*) *leD not-le-imp-less to-nat-le*)
moreover have *x* < *CARD*('*c*) **using** *x bij-to-nat*[**where** *?'a*='*b*] **unfolding** *bij-betw-def* **by** (*metis less-le-trans not-le to-nat-less-card*)
ultimately show ∃ *xa*≤*GREATEST a.* ¬ *is-zero-row a* (*Gauss-Jordan A*). *x* = *to-nat xa* **using** *to-nat-from-nat-id* **by** *fastforce*
qed
qed
also have ... = *to-nat* (*GREATEST a.* ¬ *is-zero-row a* (*Gauss-Jordan A*)) + *1* **unfolding** *card-Collect-le-nat* **by** *simp*
finally show *?thesis* **.**
qed

lemma *rank-less-row-i-imp-i-is-zero*:
assumes *rank-less-i*: *to-nat i* ≥ *rank A*
shows *Gauss-Jordan A* $ *i* = *0*
proof (*cases A=0*)
case *True* **thus** *?thesis* **by** (*metis A-0-imp-Gauss-Jordan-0 zero-index*)
next
case *False*
have *to-nat i* ≥ *to-nat* (*GREATEST a.* ¬ *is-zero-row a* (*Gauss-Jordan A*)) + *1* **using** *rank-less-i* **unfolding** *rank-eq-suc-to-nat-greatest*[*OF False*] **.**
hence *i*>(*GREATEST a.* ¬ *is-zero-row a* (*Gauss-Jordan A*))
by (*metis One-nat-def add.commute add-strict-increasing*
add-strict-increasing2 le0 lessI neq-iff not-le to-nat-mono)
hence *is-zero-row i* (*Gauss-Jordan A*) **using** *not-greater-Greatest* **by** *auto*
thus *?thesis* **unfolding** *is-zero-row-def' vec-eq-iff* **by** *auto*
qed

lemma *rank-Gauss-Jordan-eq*:
fixes *A*::'*a*::{*field*}^'*n*::{*mod-type*}^'*m*::{*mod-type*}
shows *rank A* = (*let A'*=(*Gauss-Jordan A*) *in card* {*row i A'* |*i. row i A'* ≠ *0*})
by (*metis* (*mono-tags*) *rank-Gauss-Jordan rref-Gauss-Jordan rref-rank*)

6.4 Lemmas for code generation and rank computation

lemma [*code abstract*]:
shows *vec-nth* (*Gauss-Jordan-in-ij A i j*) = (*let n* = (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*);
interchange-A = (*interchange-rows A i n*);
A' = *mult-row interchange-A i* (*1*/*interchange-A*$*i*$*j*) *in*
(% *s. if s*=*i then A'* $ *s else* (*row-add A' s i* (−(*interchange-A*$*s*$*j*))) $ *s*))
unfolding *Gauss-Jordan-in-ij-def Let-def* **by** *fastforce*

lemma *rank-Gauss-Jordan-code*[*code*]:
fixes *A*::'*a*::{*field*}^'*n*::{*mod-type*}^'*m*::{*mod-type*}
shows *rank A* = (*if A* = *0 then 0 else* (*let A'*=(*Gauss-Jordan A*) *in to-nat* (*GREATEST a. row a A'* ≠ *0*) + *1*))

proof (*cases A = 0*)
case *True* **show** *?thesis* **unfolding** *if-P*[*OF True*] **unfolding** *True rank-0* **..**
next
case *False*
show *?thesis* **unfolding** *if-not-P*[*OF False*]
unfolding *rank-eq-suc-to-nat-greatest*[*OF False*] *Let-def is-zero-row-eq-row-zero*
..
qed

lemma *dim-null-space*[*code-unfold*]:
fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
shows *vec.dim* (*null-space A*) = (*vec.dimension TYPE*(*'a*) *TYPE*(*'cols*)) − *rank* (*A*)
apply (*rule add-implies-diff*)
using *rank-nullity-theorem-matrices*
unfolding *rank-eq-dim-col-space*[*of A*]
unfolding *dimension-vector ncols-def* **..**

lemma *rank-eq-dim-col-space'*[*code-unfold*]:
fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
shows *vec.dim* (*col-space A*) = *rank A* **unfolding** *rank-eq-dim-col-space* **..**

lemma *dim-left-null-space*[*code-unfold*]:
fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
shows *vec.dim* (*left-null-space A*) = (*vec.dimension TYPE*(*'a*) *TYPE*(*'rows*)) − *rank* (*A*)
unfolding *left-null-space-eq-null-space-transpose*
unfolding *dim-null-space* **unfolding** *rank-transpose* **..**

lemmas *rank-col-rank*[*symmetric*, *code-unfold*]
lemmas *rank-def*[*symmetric*, *code-unfold*]
lemmas *row-rank-def*[*symmetric*, *code-unfold*]
lemmas *col-rank-def*[*symmetric*, *code-unfold*]
lemmas *DIM-cart*[*code-unfold*]
lemmas *DIM-real*[*code-unfold*]

end

7 Linear Maps

theory *Linear-Maps*
imports
Gauss-Jordan
begin

lemma ((λ(*x*, *y*). (*x*::*real* , − *y*::*real*)) *has-derivative* (λ*h*. (*fst h*, − *snd h*))) (*at x*)
apply (*rule has-derivative-eq-rhs*)
apply (*rule has-derivative-split*)

apply (*rule has-derivative-Pair*)
by (*auto intro*!: *derivative-eq-intros*)

7.1 Properties about ranks and linear maps

lemma *rank-matrix-dim-range*:
assumes *lf*: *linear* ((∗*s*)) ((∗*s*)) *f*
shows *rank* (*matrix f*::′*a*::{*field*}^′*cols*::{*mod-type*}^′*rows*::{*mod-type*}) = *vec.dim* (*range f*)
unfolding *rank-col-rank*[*of matrix f*] *col-rank-def*
unfolding *col-space-eq*′ **using** *matrix-works*[*OF lf*] **by** *metis*

The following two lemmas are the demonstration of theorem 2.11 that appears the book "Advanced Linear Algebra" by Steven Roman.

lemma *linear-injective-rank-eq-ncols*:
assumes *lf*: *linear* ((∗*s*)) ((∗*s*)) *f*
shows *inj f* ⟷ *rank* (*matrix f*::′*a*::{*field*}^′*cols*::{*mod-type*}^′*rows*::{*mod-type*}) = *ncols* (*matrix f*)
proof (*rule*)
interpret *lf*: *Vector-Spaces.linear* ((∗*s*)) ((∗*s*)) *f* **using** *lf* **by** *simp*
assume *inj*: *inj f*
hence {*x*. *f x* = *0*} = {*0*} **using** *lf.linear-injective-ker-0* **by** *blast*
hence *vec.dim* {*x*. *f x* = *0*} = *0* **using** *vec.dim-zero-eq*′ **by** *blast*
thus *rank* (*matrix f*) = *ncols* (*matrix f*) **using** *vec.rank-nullity-theorem* **unfolding** *ncols-def*
using *rank-matrix-dim-range*[*OF lf*]
by (*metis add.left-neutral lf.linear-axioms vec.dim-UNIV vec.dimension-def vec-dim-card*)
next
assume *eq*: *rank* (*matrix f*::′*a*::{*field*}^′*cols*::{*mod-type*}^′*rows*::{*mod-type*}) = *ncols* (*matrix f*)
have *vec.dim* {*x*. *f x* = *0*} = *0*
unfolding *ncols-def*
using *rank-matrix-dim-range*[*OF lf*] *eq*
by (*metis cancel-comm-monoid-add-class.diff-cancel dim-null-space lf ncols-def null-space-eq-ker vec.dim-UNIV vec.dimension-def vec-dim-card*)
hence {*x*. *f x* = *0*} = {*0*} **using** *vec.dim-zero-eq*
using *lf vec.linear-0* **by** *auto*
thus *inj f* **using** *vec.linear-injective-ker-0*[*of matrix f*]
unfolding *matrix-vector-mul*(*1*)[*OF lf*] **by** *simp*
qed

lemma *linear-surjective-rank-eq-ncols*:
assumes *lf*: *linear* ((∗*s*)) ((∗*s*)) *f*
shows *surj f* ⟷ *rank* (*matrix f*::′*a*::{*field*}^′*cols*::{*mod-type*}^′*rows*::{*mod-type*}) = *nrows* (*matrix f*)
proof (*rule*)
assume *surj*: *surj f*
have *nrows* (*matrix f*) = *CARD* (′*rows*) **unfolding** *nrows-def* **..**

also have ... = *vec.dim* (*range f*) **by** (*metis surj vec-dim-card*)
also have ... = *rank* (*matrix f*) **unfolding** *rank-matrix-dim-range*[*OF lf*] **..**
finally show *rank* (*matrix f*) = *nrows* (*matrix f*) **..**
next
assume *rank* (*matrix f*) = *nrows* (*matrix f*)
hence *vec.dim* (*range f*) = *CARD* ('*rows*) **unfolding** *rank-matrix-dim-range*[*OF lf*] *nrows-def* **.**
thus *surj f*
using *vec.basis-exists independent-is-basis is-basis-def lf*
vec.subspace-UNIV vec.subspace-image
by (*metis col-space-eq' col-space-eq-range vec.span-subspace*)
qed

lemma *linear-bij-rank-eq-ncols*:
fixes *f*::('*a*::{*field*}^'*n*::{*mod-type*})=>('*a*::{*field*}^'*n*::{*mod-type*})
assumes *lf*: *linear* ((∗*s*)) ((∗*s*)) *f*
shows *bij f* ⟷ *rank* (*matrix f*) = *ncols* (*matrix f*)
unfolding *bij-def*
using *lf linear-injective-rank-eq-ncols vec.linear-inj-imp-surj* **by** *auto*

7.2 Invertible linear maps

locale *invertible-lf* = *Vector-Spaces.linear* +
assumes *invertible*: (∃ *g*. *g* ∘ *f* = *id* ∧ *f* ∘ *g* = *id*)
begin

lemma *invertible-lf*: (∃ *g*. *linear* ((∗*b*)) ((∗*a*)) *g* ∧ (*g* ∘ *f* = *id*) ∧ (*f* ∘ *g* = *id*))
proof −
have *inj-on f UNIV*
using *invertible* **by** (*auto simp*: *o-def id-def inj-on-def fun-eq-iff*) *metis*
from *linear-exists-left-inverse-on*[*OF linear-axioms vs1.subspace-UNIV this*] **obtain** *g* **where**
g: *linear* (∗*b*) (∗*a*) *g g o f* = *id*
by (*auto simp*: *fun-eq-iff id-def module-hom-iff-linear*)
then have *f o g* = *id*
using *invertible*
by (*auto simp*: *inj-on-def fun-eq-iff*) *metis*
with *g* **show** *?thesis* **by** *auto*
qed

end

lemma (**in** *Vector-Spaces.linear*) *invertible-lf-intro*[*intro*]:
assumes (*g* ∘ *f* = *id*) **and** (*f* ∘ *g* = *id*)
shows *invertible-lf* ((∗*a*)) ((∗*b*)) *f*
using *assms*
by *unfold-locales auto*

lemma *invertible-imp-bijective*:

assumes *invertible-lf scaleB scaleC f*
shows *bij f*
using *assms* **unfolding** *invertible-lf-def invertible-lf-axioms-def*
by (*metis bij-betw-comp-iff bij-betw-imp-surj inj-on-imageI2 inj-on-imp-bij-betw inv-id surj-id surj-imp-inj-inv*)

lemma *invertible-matrix-imp-invertible-lf*:
fixes *A*::*'a*::{*field*}^*'n*^*n*
assumes *invertible-A*: *invertible A*
shows *invertible-lf* ((∗*s*)) ((∗*s*)) (λ*x*. *A* ∗*v x*)
proof −
obtain *B* **where** *AB*: *A*∗∗*B*=*mat 1* **and** *BA*: *B*∗∗*A*=*mat 1* **using** *invertible-A* **unfolding** *invertible-def* **by** *blast*
show *?thesis*
proof (*rule vec.invertible-lf-intro* [*of* (λ*x*. *B* ∗*v x*)])
show *id1*: (∗*v*) *B* ∘ (∗*v*) *A* = *id* **by** (*metis* (*opaque-lifting*, *no-types*) *AB BA isomorphism-expand matrix-vector-mul-assoc matrix-vector-mul-lid*)
show (∗*v*) *A* ∘ (∗*v*) *B* = *id* **by** (*metis* (*opaque-lifting*, *no-types*) *AB BA isomorphism-expand matrix-vector-mul-assoc matrix-vector-mul-lid*)
qed
qed

lemma *invertible-lf-imp-invertible-matrix*:
fixes *f*::*'a*::{*field*}^*'n*⇒*'a*^*'n*
assumes *invertible-f*: *invertible-lf* ((∗*s*)) ((∗*s*)) *f*
shows *invertible* (*matrix f*)
proof −
interpret *i*: *invertible-lf* ((∗*s*)) ((∗*s*)) *f* **using** *invertible-f* **.**
obtain *g* **where** *linear-g*: *linear* ((∗*s*)) ((∗*s*)) *g* **and** *gf*: (*g* ∘ *f* = *id*) **and** *fg*: (*f* ∘ *g* = *id*)
by (*metis invertible-f invertible-lf.invertible-lf*)
show *?thesis*
proof (*unfold invertible-def*, *rule exI*[*of - matrix g*], *rule conjI*)
show *matrix f* ∗∗ *matrix g* = *mat 1*
unfolding *matrix-eq matrix-vector-mul-assoc*[*symmetric*]
by (*metis i.linear-axioms matrix-vector-mul-lid pointfree-idE fg linear-g matrix-vector-mul*)
show *matrix g* ∗∗ *matrix f* = *mat 1*
by (*metis* ‹*matrix f* ∗∗ *matrix g* = *mat 1*› *matrix-left-right-inverse*)
qed
qed

lemma *invertible-matrix-iff-invertible-lf*:
fixes *A*::*'a*::{*field*}^*'n*^*n*
shows *invertible A* ⟷ *invertible-lf* ((∗*s*)) ((∗*s*)) (λ*x*. *A* ∗*v x*)
by (*metis invertible-lf-imp-invertible-matrix invertible-matrix-imp-invertible-lf matrix-of-matrix-vector-mul*)

lemma *invertible-matrix-iff-invertible-lf'*:

fixes $f::'a::\{field\}\hat{}'n \Rightarrow 'a\hat{}'n$
assumes *linear-f*: *linear* $((*s))$ $((*s))$ f
shows *invertible* $(matrix\ f) \longleftrightarrow$ *invertible-lf* $((*s))$ $((*s))$ f
by (*metis* (*lifting*) *assms invertible-matrix-iff-invertible-lf matrix-vector-mul*)

lemma *invertible-matrix-mult-right-rank*:
fixes $A::'a::\{field\}\hat{}'n::\{mod\text{-}type\}\hat{}'m::\{mod\text{-}type\}$
and $Q::'a::\{field\}\hat{}'n::\{mod\text{-}type\}\hat{}'n::\{mod\text{-}type\}$
assumes *invertible-Q*: *invertible Q*
shows *rank* $(A**Q) =$ *rank A*
proof −
define *TQ* **where** $TQ\ x = Q *v\ x$ **for** x
define *TA* **where** $TA\ x = A *v\ x$ **for** x
define *TAQ* **where** $TAQ\ x = (A**Q) *v\ x$ **for** x
have *invertible-lf* $((*s))$ $((*s))$ *TQ* **using** *invertible-matrix-imp-invertible-lf*[*OF invertible-Q*] **unfolding** *TQ-def* **.**
hence *bij-TQ*: *bij TQ* **using** *invertible-imp-bijective* **by** *auto*
have *range TAQ* = *range* $(TA \circ TQ)$ **unfolding** *TQ-def TA-def TAQ-def o-def matrix-vector-mul-assoc* **..**
also have ... = *TA* ‘(*range TQ*) **unfolding** *fun.set-map* **..**
also have ... = *TA* ‘ (*UNIV*) **using** *bij-is-surj*[*OF bij-TQ*] **by** *simp*
finally have *range TAQ* = *range TA* **.**
thus *?thesis* **unfolding** *rank-eq-dim-image* **using** *TAQ-def* [*abs-def*] *TA-def* [*abs-def*] **by** *auto*
qed

lemma *subspace-image-invertible-mat*:
fixes $P::'a::\{field\}\hat{}'m\hat{}'m$
assumes *inv-P*: *invertible P*
and *sub-W*: *vec.subspace W*
shows *vec.subspace* $((\lambda x.\ P *v\ x)`\ W)$
using *assms* **by** (*intro vec.subspace-image*)

lemma *dim-image-invertible-mat*:
fixes $P::'a::\{field\}\hat{}'m\hat{}'m$
assumes *inv-P*: *invertible P*
and *sub-W*: *vec.subspace W*
shows *vec.dim* $((\lambda x.\ P *v\ x)`\ W) =$ *vec.dim W*
proof −
obtain *B* **where** *B-in-W*: $B \subseteq W$ **and** *ind-B*: *vec.independent B*
and *W-in-span-B*: $W \subseteq$ *vec.span B* **and** *card-B-eq-dim-W*: *card B* = *vec.dim W*
using *vec.basis-exists* **by** *blast*
define *L* **where** $L = (\lambda x.\ P *v\ x)$
define *C* **where** $C = L`B$
have *finite-B*: *finite B* **using** *vec.indep-card-eq-dim-span*[*OF ind-B*] **by** *simp*
interpret *L*: *Vector-Spaces.linear* $(*s)$ $(*s)$ *L* **using** *matrix-vector-mul-linear-gen* **unfolding** *L-def* **.**

have *finite-C*: *finite C* **using** *vec.indep-card-eq-dim-span*[*OF ind-B*] **unfolding** *C-def* **by** *simp*
have *inv-TP*: *invertible-lf* ((∗*s*)) ((∗*s*)) (λ*x*. *P* ∗*v x*) **using** *invertible-matrix-imp-invertible-lf*[*OF inv-P*] **.**
have *inj-on-LW*: *inj-on L W* **using** *invertible-imp-bijective*[*OF inv-TP*] **unfolding** *bij-def L-def* **unfolding** *inj-on-def*
by *blast*
hence *inj-on-LB*: *inj-on L B* **unfolding** *inj-on-def* **using** *B-in-W* **by** *auto*
have *ind-D*: *vec.independent C*
proof (*rule vec.independent-if-scalars-zero*[*OF finite-C*])
fix *f x*
assume *sum*: (∑ *x*∈*C*. *f x* ∗*s x*) = *0* **and** *x*: *x* ∈ *C*
obtain *y* **where** *Ly-eq-x*: *L y* = *x* **and** *y*: *y* ∈ *B* **using** *x* **unfolding** *C-def L-def* **by** *auto*
have (∑ *x*∈*C*. *f x* ∗*s x*) = *sum* ((λ*x*. *f x* ∗*s x*) ∘ *L*) *B* **unfolding** *C-def* **by** (*rule sum.reindex*[*OF inj-on-LB*])
also have ... = *sum* (λ*x*. *f* (*L x*) ∗*s L x*) *B* **unfolding** *o-def* **..**
also have ... = *sum* (λ*x*. ((*f* ∘ *L*) *x*) ∗*s L x*) *B* **using** *o-def* **by** *auto*
also have ... = *L* (*sum* (λ*x*. ((*f* ∘ *L*) *x*) ∗*s x*) *B*)
by (*simp add*: *L.sum L.scale*)
finally have *rw*: (∑ *x*∈*C*. *f x* ∗*s x*) = *L* (∑ *x*∈*B*. (*f* ∘ *L*) *x* ∗*s x*) **.**
have (∑ *x*∈*B*. (*f* ∘ *L*) *x* ∗*s x*) ∈ *W*
by (*rule vec.subspace-sum*[*OF sub-W*])
(*simp add*: *B-in-W rev-subsetD sub-W vec.subspace-scale*)
hence (∑ *x*∈*B*. (*f* ∘ *L*) *x* ∗*s x*)=*0*
using *sum rw*
using *vec.linear-inj-on-iff-eq-0*[*OF L.linear-axioms sub-W*] **using** *inj-on-LW*
by (*auto simp*: *L-def*)
hence (*f* ∘ *L*) *y* = *0*
using *vec.scalars-zero-if-independent*[*OF finite-B ind-B*, *of* (*f* ∘ *L*)]
using *y* **by** *auto*
thus *f x* = *0* **unfolding** *o-def Ly-eq-x* **.**
qed
have *L‘ W* ⊆ *vec.span C*
proof (*unfold vec.span-finite*[*OF finite-C*], *clarify*)
fix *xa* **assume** *xa-in-W*: *xa* ∈ *W*
obtain *g* **where** *sum-g*: *sum* (λ*x*. *g x* ∗*s x*) *B* = *xa*
using *vec.span-finite*[*OF finite-B*] *W-in-span-B xa-in-W* **by** *force*
show *L xa* ∈ *range* (λ*u*. (∑ *v*∈*C*. *u v* ∗*s v*))
proof (*rule image-eqI*[**where** *x*=λ*x*. *g* (*THE y*. *y* ∈ *B* ∧ *x*=*L y*)])
have *L xa* = *L* (*sum* (λ*x*. *g x* ∗*s x*) *B*) **using** *sum-g* **by** *simp*
also have ... = *sum* (λ*x*. *g x* ∗*s L x*) *B* **by** (*simp add*: *L.sum L.scale*)
also have ... = *sum* (λ*x*. *g* (*THE y*. *y* ∈ *B* ∧ *x*=*L y*) ∗*s x*) (*L‘B*)
proof (*unfold sum.reindex*[*OF inj-on-LB*], *unfold o-def*, *rule sum.cong*)
fix *x* **assume** *x-in-B*: *x*∈*B*
have *x-eq-the*:*x* = (*THE y*. *y* ∈ *B* ∧ *L x* = *L y*)
proof (*rule the-equality*[*symmetric*])
show *x* ∈ *B* ∧ *L x* = *L x* **using** *x-in-B* **by** *auto*
show ⋀*y*. *y* ∈ *B* ∧ *L x* = *L y* ⟹ *y* = *x* **using** *inj-on-LB x-in-B* **unfolding**

inj-on-def **by** *fast*
qed
show $g\ x * s\ L\ x = g\ (THE\ y.\ y \in B \wedge L\ x = L\ y) * s\ L\ x$ **using** *x-eq-the*
by *simp*
qed *rule*
finally show $L\ xa = (\sum v \in C.\ g\ (THE\ y.\ y \in B \wedge v = L\ y) * s\ v)$ **unfolding**
C-def **by** *simp*
qed *rule*
qed
have *card C* = *card B* **using** *card-image*[*OF inj-on-LB*] **unfolding** *C-def* **.**
thus *?thesis*
by (*metis B-in-W C-def L.span-image W-in-span-B* ‹$L \equiv (*v)\ P$› *card-B-eq-dim-W*
ind-D sub-W
vec.indep-card-eq-dim-span vec.span-subspace)
qed

lemma *invertible-matrix-mult-left-rank*:
fixes *A*::'*a*::{*field*}^'*n*::{*mod-type*}^'*m*::{*mod-type*}
and *P*::'*a*::{*field*}^'*m*::{*mod-type*}^'*m*::{*mod-type*}
assumes *invertible-P*: *invertible P*
shows *rank* (*P***A*) = *rank A*
proof −
define *TP* **where** *TP* = (λ*x*. *P* ∗*v x*)
define *TA* **where** *TA* = (λ*x*. *A* ∗*v x*)
define *TPA* **where** *TPA* = (λ*x*. (*P***A*) ∗*v x*)
have *sub*: *vec.subspace* (*range* ((∗*v*) *A*))
by (*metis vec.subspace-UNIV vec.subspace-image*)
have *vec.dim* (*range TPA*) = *vec.dim* (*range* (*TP* ∘ *TA*))
unfolding *TP-def TA-def TPA-def o-def matrix-vector-mul-assoc* **..**
also have ... = *vec.dim* (*range TA*) **using** *dim-image-invertible-mat*[*OF invert-*
ible-P sub]
unfolding *TP-def TA-def o-def fun.set-map*[*symmetric*] **.**
finally show *?thesis* **unfolding** *rank-eq-dim-image TPA-def TA-def* **.**
qed

corollary *invertible-matrices-mult-rank*:
fixes *A*::'*a*::{*field*}^'*n*::{*mod-type*}^'*m*::{*mod-type*}
and *P*::'*a*^'*m*::{*mod-type*}^'*m*::{*mod-type*} **and** *Q*::'*a*^'*n*::{*mod-type*}^'*n*::{*mod-type*}
assumes *invertible-P*: *invertible P*
and *invertible-Q*: *invertible Q*
shows *rank* (*P***A***Q*) = *rank A*
using *invertible-matrix-mult-right-rank*[*OF invertible-Q*] **using** *invertible-matrix-mult-left-rank*[*OF*
invertible-P] **by** *metis*

lemma *invertible-matrix-mult-left-rank'*:
fixes *A*::'*a*::{*field*}^'*n*::{*mod-type*}^'*m*::{*mod-type*} **and** *P*::'*a*^'*m*::{*mod-type*}^'*m*::{*mod-type*}
assumes *invertible-P*: *invertible P* **and** *B-eq-PA*: *B*=*P***A*

shows *rank B = rank A*
proof −
have *rank B = rank (P∗∗A)* **using** *B-eq-PA* **by** *auto*
also have *... = rank A* **using** *invertible-matrix-mult-left-rank*[*OF invertible-P*]
by *auto*
finally show *?thesis* **.**
qed

lemma *invertible-matrix-mult-right-rank′*:
fixes *A*::*′a*::{*field*}^*′n*::{*mod-type*}^*′m*::{*mod-type*}
and *Q*::*′a*::{*field*}^*′n*::{*mod-type*}^*′n*::{*mod-type*}
assumes *invertible-Q*: *invertible Q* **and** *B-eq-PA*: *B=A∗∗Q*
shows *rank B = rank A* **by** (*metis B-eq-PA invertible-Q invertible-matrix-mult-right-rank*)

lemma *invertible-matrices-rank′*:
fixes *A*::*′a*::{*field*}^*′n*::{*mod-type*}^*′m*::{*mod-type*}
and *P*::*′a*^*′m*::{*mod-type*}^*′m*::{*mod-type*} **and** *Q*::*′a*^*′n*::{*mod-type*}^*′n*::{*mod-type*}
assumes *invertible-P*: *invertible P* **and** *invertible-Q*: *invertible Q* **and** *B-eq-PA*: *B = P∗∗A∗∗Q*
shows *rank B = rank A* **by** (*metis B-eq-PA invertible-P invertible-Q invertible-matrices-mult-rank*)

7.3 Definition and properties of the set of a vector

Some definitions:

In the file *Generalizations.thy* there exists the following definition: *cart-basis = {axis i 1 |i. i ∈ UNIV}*.

cart-basis returns a set which is a basis and it works properly in my development. But in this file, I need to know the order of the elements of the basis, because is very important for the coordenates of a vector and the matrices of change of bases. So, I have defined a new *cart-basis′*, which will be a matrix. The columns of this matrix are the elements of the basis.

definition *set-of-vector* :: *′a^*′n ⇒ *′a set*
where *set-of-vector A = {A $ i |i. i ∈ UNIV}*

definition *cart-basis′* :: *′a*::{*field*}^*′n^*′n*
where *cart-basis′ = (χ i. axis i 1)*

lemma *cart-basis-eq-set-of-vector-cart-basis′*:
cart-basis = set-of-vector (*cart-basis′*)
unfolding *cart-basis-def cart-basis′-def set-of-vector-def* **by** *auto*

lemma *basis-image-linear*:
fixes *f*::*′b*::{*field*}^*′n => ′b^*′n*
assumes *invertible-lf*: *invertible-lf* ((∗*s*)) ((∗*s*)) *f*
and *basis-X*: *is-basis* (*set-of-vector X*)
shows *is-basis* (*f‘* (*set-of-vector X*))

proof (*rule iffD1*[*OF independent-is-basis*], *rule conjI*)
have *card* (*f ‘ set-of-vector X*) = *card* (*set-of-vector X*)
by (*rule card-image*[*of f set-of-vector X*], *metis invertible-imp-bijective*[*OF invertible-lf*] *bij-def inj-eq inj-on-def*)
also have ... = *card* (*UNIV*::′*n set*) **using** *basis-X* **unfolding** *independent-is-basis*[*symmetric*]
by *auto*
finally show *card* (*f ‘ set-of-vector X*) = *card* (*UNIV*::′*n set*) **.**
interpret *Vector-Spaces.linear* (∗*s*) (∗*s*) *f* **using** *invertible-lf* **unfolding** *invertible-lf-def* **by** *simp*
show *vec.independent* (*f ‘ set-of-vector X*)
proof (*rule independent-injective-image*)
show *vec.independent* (*set-of-vector X*) **using** *basis-X* **unfolding** *is-basis-def*
by *simp*
show *inj-on f* (*vec.span* (*set-of-vector X*))
by (*metis bij-def injD inj-onI invertible-imp-bijective invertible-lf*)
qed
qed

Properties about *cart-basis′* = (*χi. axis i 1*)

lemma *set-of-vector-cart-basis′*:
shows (*set-of-vector cart-basis′*) = {*axis i 1* :: ′*a*::{*field*}^′*n* | *i. i* ∈ (*UNIV* :: ′*n set*)}
unfolding *set-of-vector-def cart-basis′-def* **by** *auto*

lemma *cart-basis′-i*: *cart-basis′* $ *i* = *axis i 1* **unfolding** *cart-basis′-def* **by** *simp*

lemma *finite-set-of-vector*[*intro*,*simp*]: *finite* (*set-of-vector X*)
unfolding *set-of-vector-def* **using** *finite-Atleast-Atmost-nat*[*of* λ*i. X* $ *i*] **.**

lemma *is-basis-cart-basis′*: *is-basis* (*set-of-vector cart-basis′*)
unfolding *cart-basis-eq-set-of-vector-cart-basis′*[*symmetric*]
by (*metis Miscellaneous.is-basis-def independent-cart-basis span-cart-basis*)

lemma *basis-expansion-cart-basis′*:*sum* (λ*i. x*$*i* ∗*s cart-basis′* $ *i*) *UNIV* = *x*
unfolding *cart-basis′-def* **using** *basis-expansion* **by** *auto*

lemma *basis-expansion-unique*:
sum (λ*i. f i* ∗*s axis* (*i*::′*n*::*finite*) *1*) *UNIV* = (*x*::(′*a*::*comm-ring-1*) ^′*n*) ⟷ (∀ *i. f i* = *x*$*i*)
proof (*auto simp add*: *basis-expansion*)
fix *i*::′*n*
have *univ-rw*: *UNIV* = (*UNIV* − {*i*}) ∪ {*i*} **by** *fastforce*
have (∑ *x*∈*UNIV*. *f x* ∗ *axis x 1* $ *i*) = *sum* (λ*x*. *f x* ∗ *axis x 1* $ *i*) (*UNIV* − {*i*} ∪ {*i*}) **using** *univ-rw* **by** *simp*
also have ... = *sum* (λ*x*. *f x* ∗ *axis x 1* $ *i*) (*UNIV* − {*i*}) + *sum* (λ*x*. *f x* ∗ *axis x 1* $ *i*) {*i*} **by** (*rule sum.union-disjoint*, *auto*)
also have ... = *f i* **unfolding** *axis-def* **by** *auto*
finally show *f i* = (∑ *x*∈*UNIV*. *f x* ∗ *axis x 1* $ *i*) **..**
qed

lemma *basis-expansion-cart-basis′-unique*: *sum* (λ*i*. *f* (*cart-basis′* $ *i*) ∗*s* *cart-basis′* $ *i*) *UNIV* = *x* ⟷ (∀ *i*. *f* (*cart-basis′* $ *i*) = *x*$*i*)
using *basis-expansion-unique* **unfolding** *cart-basis′-def*
by (*simp add*: *vec-eq-iff if-distrib cong del*: *if-weak-cong*)

lemma *basis-expansion-cart-basis′-unique′*: *sum* (λ*i*. *f i* ∗*s* *cart-basis′* $ *i*) *UNIV* = *x* ⟷ (∀ *i*. *f i* = *x*$*i*)
using *basis-expansion-unique* **unfolding** *cart-basis′-def*
by (*simp add*: *vec-eq-iff if-distrib cong del*: *if-weak-cong*)

Properties of *is-basis ?S* ≡ *vec.independent ?S* ∧ *vec.span ?S* = *UNIV*.

lemma *sum-basis-eq*:
fixes *X*::′*a*::{*field*}^′*n*^′*n*
assumes *is-basis*:*is-basis* (*set-of-vector X*)
shows *sum* (λ*x*. *f x* ∗*s x*) (*set-of-vector X*) = *sum* (λ*i*. *f* (*X*$*i*) ∗*s* (*X*$*i*)) *UNIV*
proof −
have *card-set-of-vector*:*card*(*set-of-vector X*) = *CARD*(′*n*)
using *independent-is-basis*[*of set-of-vector X*] **using** *is-basis* **by** *auto*
have *fact-1*: *set-of-vector X* = *range* (($) *X*) **unfolding** *set-of-vector-def* **by** *auto*
have *inj*: *inj* (($) *X*)
proof (*rule eq-card-imp-inj-on*)
show *finite* (*UNIV*::′*n set*) **using** *finite-class.finite-UNIV* .
show *card* (*range* (($) *X*)) = *card* (*UNIV*::′*n set*)
using *card-set-of-vector* **using** *fact-1* **unfolding** *set-of-vector-def* **by** *simp*
qed
show *?thesis* **using** *sum.reindex*[*OF inj*, *of* (λ*x*. *f x* ∗*s x*), *unfolded o-def*] **unfolding** *fact-1* .
qed

corollary *sum-basis-eq2*:
fixes *X*::′*a*::{*field*}^′*n*^′*n*
assumes *is-basis*:*is-basis* (*set-of-vector X*)
shows *sum* (λ*x*. *f x* ∗*s x*) (*set-of-vector X*) = *sum* (λ*i*. (*f* ∘ ($) *X*) *i* ∗*s* (*X*$*i*)) *UNIV*
using *sum-basis-eq*[*OF is-basis*] **by** *simp*

lemma *inj-op-nth*:
fixes *X*::′*a*::{*field*}^′*n*^′*n*
assumes *is-basis*: *is-basis* (*set-of-vector X*)
shows *inj* (($) *X*)
proof −
have *fact-1*: *set-of-vector X* = *range* (($) *X*) **unfolding** *set-of-vector-def* **by** *auto*
have *card-set-of-vector*:*card*(*set-of-vector X*) = *CARD*(′*n*) **using** *independent-is-basis*[*of set-of-vector X*] **using** *is-basis* **by** *auto*
show *inj* (($) *X*)
proof (*rule eq-card-imp-inj-on*)

show *finite* (*UNIV*::′*n set*) **using** *finite-class.finite-UNIV* **.**
show *card* (*range* ((\$) *X*)) = *card* (*UNIV*::′*n set*) **using** *card-set-of-vector* **using** *fact-1* **unfolding** *set-of-vector-def* **by** *simp*
qed
qed

lemma *basis-UNIV*:
fixes *X*::′*a*::{*field*}^′*n*^′*n*
assumes *is-basis*: *is-basis* (*set-of-vector X*)
shows *UNIV* = {*x*. ∃ *g*. (∑ *i*∈*UNIV*. *g i* ∗*s X*\$*i*) = *x*}
proof −
have *UNIV* = {*x*. ∃ *g*. (∑ *i*∈(*set-of-vector X*). *g i* ∗*s i*) = *x*}
using *vec.span-finite*[*OF basis-finite*[*OF is-basis*]]
using *is-basis* **unfolding** *is-basis-def*
by (*auto simp*: *set-eq-iff*
intro!: *vec.sum-representation-eq exI*[**where** *x*=*vec.representation* (*set-of-vector X*) *x* **for** *x*])
also have ... ⊆ {*x*. ∃ *g*. (∑ *i*∈*UNIV*. *g i* ∗*s X*\$*i*) = *x*}
proof (*clarify*)
fix *f*
show ∃ *g*. (∑ *i*∈*UNIV*. *g i* ∗*s X* \$ *i*) = (∑ *i*∈*set-of-vector X*. *f i* ∗*s i*)
proof (*rule exI*[*of* - (λ*i*. (*f* ∘ (\$) *X*) *i*)], *unfold o-def*)
have *fact-1*: *set-of-vector X* = *range* ((\$) *X*) **unfolding** *set-of-vector-def* **by** *auto*
have *card-set-of-vector*:*card*(*set-of-vector X*) = *CARD*(′*n*) **using** *independent-is-basis*[*of set-of-vector X*] **using** *is-basis* **by** *auto*
have *inj*: *inj* ((\$) *X*) **using** *inj-op-nth*[*OF is-basis*] **.**
show (∑ *i*∈*UNIV*. *f* (*X* \$ *i*) ∗*s X* \$ *i*) = (∑ *i*∈*set-of-vector X*. *f i* ∗*s i*)
using *sum.reindex*[*symmetric*, *OF inj*, *of* λ*i*. *f i* ∗*s i*] **unfolding** *fact-1* **by** *simp*
qed
qed
finally show *?thesis* **by** *auto*
qed

lemma *scalars-zero-if-basis*:
fixes *X*::′*a*::{*field*}^′*n*^′*n*
assumes *is-basis*: *is-basis* (*set-of-vector X*) **and** *sum*: (∑ *i*∈(*UNIV*::′*n set*). *f i* ∗*s X*\$*i*) = *0*
shows ∀ *i*∈(*UNIV*::′*n set*). *f i* = *0*
proof −
have *ind-X*: *vec.independent* (*set-of-vector X*) **using** *is-basis* **unfolding** *is-basis-def* **by** *simp*
have *finite-X*:*finite* (*set-of-vector X*) **using** *basis-finite*[*OF is-basis*] **.**
have *1*: (∀ *g*. (∑ *v*∈(*set-of-vector X*). *g v* ∗*s v*) = *0* ⟶ (∀ *v*∈(*set-of-vector X*). *g v* = *0*))
using *ind-X* **unfolding** *vec.independent-explicit* **using** *finite-X* **by** *auto*
define *g* **where** *g v* = *f* (*THE i*. *X* \$ *i* = *v*) **for** *v*
have (∑ *v*∈(*set-of-vector X*). *g v* ∗*s v*) = *0*

```
  proof −
    have (∑ v∈(set-of-vector X). g v ∗s v)  = (∑ i∈(UNIV::'n set). f i ∗s X$i)
    proof −
      have inj: inj (($) X) using inj-op-nth[OF is-basis] .
       have rw: set-of-vector X = range (($) X) unfolding set-of-vector-def by
auto
      {
       fix a
       have f a = g (X $ a)
         unfolding g-def using inj-op-nth[OF is-basis]
         by (metis (lifting, mono-tags) injD the-equality)
       }
      thus ?thesis using sum.reindex[OF inj, of λv. g v ∗s v] unfolding rw o-def
by auto
    qed
    thus ?thesis unfolding sum .
  qed
  hence 2: ∀ v∈(set-of-vector X). g v = 0 using 1 by auto
  show ?thesis
  proof (clarify)
    fix a
    have g (X$a) = 0 using 2 unfolding set-of-vector-def by auto
    thus f a = 0 unfolding g-def using inj-op-nth[OF is-basis]
      by (metis (lifting, mono-tags) injD the-equality)
  qed
qed

lemma basis-combination-unique:
  fixes X::'a::{field}^'n^'n
   assumes basis-X: is-basis (set-of-vector X) and sum-eq: (∑ i∈UNIV. g i ∗s
X$i) = (∑ i∈UNIV. f i ∗s X$i)
  shows f=g
proof (rule ccontr)
  assume f≠g
  from this obtain x where fx-gx: f x ≠ g x by fast
  have 0=(∑ i∈UNIV. g i ∗s X$i) − (∑ i∈UNIV. f i ∗s X$i) using sum-eq by
simp
 also have ... = (∑ i∈UNIV. g i ∗s X$i − f i ∗s X$i) unfolding sum-subtractf[symmetric]
..
  also have ... = (∑ i∈UNIV. (g i − f i) ∗s X$i)  by (rule sum.cong, auto simp
add: scaleR-diff-left)
  also have ... = (∑ i∈UNIV. (g − f) i ∗s X$i) by simp
  finally have sum-eq-1: 0 = (∑ i∈UNIV. (g − f) i ∗s X$i) by simp
 have ∀ i∈UNIV. (g − f) i = 0 by (rule scalars-zero-if-basis[OF basis-X sum-eq-1[symmetric]])
  hence (g − f) x = 0 by simp
  hence f x = g x by simp
  thus False using fx-gx by contradiction
qed
```

7.4 Coordinates of a vector

Definition and properties of the coordinates of a vector (in terms of a particular ordered basis).

definition *coord* :: *'a*::{*field*}^'*n*^'*n*⇒*'a*::{*field*}^'*n*⇒*'a*::{*field*}^'*n*
where *coord X v* = (*χ i.* (*THE f.* *v* = *sum* (*λx.* *f x* ∗*s* *X*$*x*) *UNIV*) *i*)

coord X v are the coordinates of vector *v* with respect to the basis *X*

lemma *bij-coord*:
fixes *X*::*'a*::{*field*}^'*n*^'*n*
assumes *basis-X*: *is-basis* (*set-of-vector X*)
shows *bij* (*coord X*)
proof (*unfold bij-def*, *auto*)
show *inj*: *inj* (*coord X*)
proof (*unfold inj-on-def*, *auto*)
fix *x y* **assume** *coord-eq*: *coord X x* = *coord X y*
obtain *f* **where** *f*: (∑ *x*∈*UNIV*. *f x* ∗*s* *X* $ *x*) = *x* **using** *basis-UNIV*[*OF basis-X*] **by** *blast*
obtain *g* **where** *g*: (∑ *x*∈*UNIV*. *g x* ∗*s* *X* $ *x*) = *y* **using** *basis-UNIV*[*OF basis-X*] **by** *blast*
have *the-f*: (*THE f.* *x* = (∑ *x*∈*UNIV*. *f x* ∗*s* *X* $ *x*)) = *f*
proof (*rule the-equality*)
show *x* = (∑ *x*∈*UNIV*. *f x* ∗*s* *X* $ *x*) **using** *f* **by** *simp*
show ⋀*fa*. *x* = (∑ *x*∈*UNIV*. *fa x* ∗*s* *X* $ *x*) ⟹ *fa* = *f* **using** *basis-combination-unique*[*OF basis-X*] *f* **by** *simp*
qed
have *the-g*: (*THE g.* *y* = (∑ *x*∈*UNIV*. *g x* ∗*s* *X* $ *x*)) = *g*
proof (*rule the-equality*)
show *y* = (∑ *x*∈*UNIV*. *g x* ∗*s* *X* $ *x*) **using** *g* **by** *simp*
show ⋀*ga*. *y* = (∑ *x*∈*UNIV*. *ga x* ∗*s* *X* $ *x*) ⟹ *ga* = *g* **using** *basis-combination-unique*[*OF basis-X*] *g* **by** *simp*
qed
have (*THE f.* *x* = (∑ *x*∈*UNIV*. *f x* ∗*s* *X* $ *x*)) = (*THE g.* *y* = (∑ *x*∈*UNIV*. *g x* ∗*s* *X* $ *x*))
using *coord-eq* **unfolding** *coord-def*
using *vec-lambda-inject*[*of* (*THE f.* *x* = (∑ *x*∈*UNIV*. *f x* ∗*s* *X* $ *x*)) (*THE f.* *y* = (∑ *x*∈*UNIV*. *f x* ∗*s* *X* $ *x*))]
by *auto*
hence *f* = *g* **unfolding** *the-f the-g* **.**
thus *x*=*y* **using** *f g* **by** *simp*
qed
next
fix *x*::('*a*, '*n*) *vec*
show *x* ∈ *range* (*coord X*)
proof (*unfold image-def*, *auto*, *rule exI*[*of* - *sum* (*λi.* *x*$*i* ∗*s* *X*$*i*) *UNIV*], *unfold coord-def*)
define *f* **where** *f i* = *x*$*i* **for** *i*
have *the-f*: (*THE f.* (∑ *i*∈*UNIV*. *x* $ *i* ∗*s* *X* $ *i*) = (∑ *x*∈*UNIV*. *f x* ∗*s* *X* $ *x*)) = *f*

proof (*rule the-equality*)
show $(\sum i{\in}UNIV.\ x\ \$\ i *s X\ \$\ i) = (\sum x{\in}UNIV.\ f\ x *s X\ \$\ x)$ **unfolding** *f-def* **..**
fix *g* **assume** *sum-eq*:$(\sum i{\in}UNIV.\ x\ \$\ i *s X\ \$\ i) = (\sum x{\in}UNIV.\ g\ x *s X\ \$\ x)$
show $g = f$ **using** *basis-combination-unique*[*OF basis-X*] **using** *sum-eq* **unfolding** *f-def* **by** *simp*
qed
show $x =$ *vec-lambda* $(THE\ f.\ (\sum i{\in}UNIV.\ x\ \$\ i *s X\ \$\ i) = (\sum x{\in}UNIV.\ f\ x *s X\ \$\ x))$ **unfolding** *the-f* **unfolding** *f-def* **using** *vec-lambda-eta*[*of x*] **by** *simp*
qed
qed

lemma *linear-coord*:
fixes $X::'a::\{field\}\hat{}'n\hat{}'n$
assumes *basis-X*: *is-basis* (*set-of-vector X*)
shows *linear* $((*s))$ $((*s))$ (*coord X*)
proof *unfold-locales*
fix $x\ y::('a,\ 'n)\ vec$
show *coord X* $(x + y) =$ *coord X x* + *coord X y*
proof −
obtain *f* **where** *f*: $(\sum a{\in}(UNIV::'n\ set).\ f\ a *s X\ \$\ a) = x + y$ **using** *basis-UNIV*[*OF basis-X*] **by** *blast*
obtain *g* **where** *g*: $(\sum x{\in}UNIV.\ g\ x *s X\ \$\ x) = x$ **using** *basis-UNIV*[*OF basis-X*] **by** *blast*
obtain *h* **where** *h*: $(\sum x{\in}UNIV.\ h\ x *s X\ \$\ x) = y$ **using** *basis-UNIV*[*OF basis-X*] **by** *blast*
define *t* **where** $t\ i = g\ i + h\ i$ **for** *i*
have *the-f*: $(THE\ f.\ x + y = (\sum x{\in}UNIV.\ f\ x *s X\ \$\ x)) = f$
proof (*rule the-equality*)
show $x + y = (\sum x{\in}UNIV.\ f\ x *s X\ \$\ x)$ **using** *f* **by** *simp*
show $\bigwedge fa.\ x + y = (\sum x{\in}UNIV.\ fa\ x *s X\ \$\ x) \Longrightarrow fa = f$ **using** *basis-combination-unique*[*OF basis-X*] *f* **by** *simp*
qed
have *the-g*: $(THE\ g.\ x = (\sum x{\in}UNIV.\ g\ x *s X\ \$\ x)) = g$
proof (*rule the-equality*)
show $x = (\sum x{\in}UNIV.\ g\ x *s X\ \$\ x)$ **using** *g* **by** *simp*
show $\bigwedge ga.\ x = (\sum x{\in}UNIV.\ ga\ x *s X\ \$\ x) \Longrightarrow ga = g$ **using** *basis-combination-unique*[*OF basis-X*] *g* **by** *simp*
qed
have *the-h*: $(THE\ h.\ y = (\sum x{\in}UNIV.\ h\ x *s X\ \$\ x)) = h$
proof (*rule the-equality*)
show $y = (\sum x{\in}UNIV.\ h\ x *s X\ \$\ x)$ **using** *h* **..**
show $\bigwedge ha.\ y = (\sum x{\in}UNIV.\ ha\ x *s X\ \$\ x) \Longrightarrow ha = h$ **using** *basis-combination-unique*[*OF basis-X*] *h* **by** *simp*
qed
have $(\sum a{\in}(UNIV::'n\ set).\ f\ a *s X\ \$\ a) = (\sum x{\in}UNIV.\ g\ x *s X\ \$\ x) + (\sum x{\in}UNIV.\ h\ x *s X\ \$\ x)$ **using** *f g h* **by** *simp*
also have ... $= (\sum x{\in}UNIV.\ g\ x *s X\ \$\ x + h\ x *s X\ \$\ x)$ **unfolding**

sum.distrib[*symmetric*] **..**
also have ... = $(\sum x{\in}UNIV.\ (g\ x\ +\ h\ x)\ *s\ X\ \$\ x)$ **by** (*rule sum.cong*, *auto simp add*: *scaleR-left-distrib*)
also have ... = $(\sum x{\in}UNIV.\ t\ x\ *s\ X\ \$\ x)$ **unfolding** *t-def* **..**
finally have $(\sum a{\in}UNIV.\ f\ a\ *s\ X\ \$\ a) = (\sum x{\in}UNIV.\ t\ x\ *s\ X\ \$\ x)$ **.**
hence *f*=*t* **using** *basis-combination-unique*[*OF basis-X*] **by** *auto*
thus *?thesis*
by (*unfold coord-def the-f the-g the-h*, *vector*, *auto*, *unfold f g h t-def*, *simp*)
qed
next
fix *c*::*'a* **and** *x*::*'a*^*'n*
show *coord X* (*c* ∗*s x*) = *c* ∗*s coord X x*
proof −
obtain *f* **where** *f*: $(\sum x{\in}UNIV.\ f\ x\ *s\ X\ \$\ x) = c\ *s\ x$ **using** *basis-UNIV*[*OF basis-X*] **by** *blast*
obtain *g* **where** *g*: $(\sum x{\in}UNIV.\ g\ x\ *s\ X\ \$\ x) = x$ **using** *basis-UNIV*[*OF basis-X*] **by** *blast*
define *t* **where** *t i* = *c* ∗ *g i* **for** *i*
have *the-f*: $(THE\ f.\ c\ *s\ x = (\sum x{\in}UNIV.\ f\ x\ *s\ X\ \$\ x)) = f$
proof (*rule the-equality*)
show $c\ *s\ x = (\sum x{\in}UNIV.\ f\ x\ *s\ X\ \$\ x)$ **using** *f* **..**
show $\bigwedge fa.\ c\ *s\ x = (\sum x{\in}UNIV.\ fa\ x\ *s\ X\ \$\ x) \Longrightarrow fa = f$ **using** *basis-combination-unique*[*OF basis-X*] *f* **by** *simp*
qed
have *the-g*: $(THE\ g.\ x = (\sum x{\in}UNIV.\ g\ x\ *s\ X\ \$\ x)) = g$**proof** (*rule the-equality*)
show $x = (\sum x{\in}UNIV.\ g\ x\ *s\ X\ \$\ x)$ **using** *g* **..**
show $\bigwedge ga.\ x = (\sum x{\in}UNIV.\ ga\ x\ *s\ X\ \$\ x) \Longrightarrow ga = g$ **using** *basis-combination-unique*[*OF basis-X*] *g* **by** *simp*
qed
have $(\sum x{\in}UNIV.\ f\ x\ *s\ X\ \$\ x) = c\ *s\ (\sum x{\in}UNIV.\ g\ x\ *s\ X\ \$\ x)$ **using** *f g* **by** *simp*
also have ... = $(\sum x{\in}UNIV.\ c\ *s\ (g\ x\ *s\ X\ \$\ x))$ **by** (*rule vec.scale-sum-right*)
also have ... = $(\sum x{\in}UNIV.\ t\ x\ *s\ X\ \$\ x)$ **unfolding** *t-def* **by** *simp*
finally have $(\sum x{\in}UNIV.\ f\ x\ *s\ X\ \$\ x) = (\sum x{\in}UNIV.\ t\ x\ *s\ X\ \$\ x)$ **.**
hence *f*=*t* **using** *basis-combination-unique*[*OF basis-X*] **by** *auto*
thus *?thesis*
by (*unfold coord-def the-f the-g*, *vector*, *auto*, *unfold t-def*, *auto*)
qed
qed

lemma *coord-eq*:
assumes *basis-X*:*is-basis* (*set-of-vector X*)
and *coord-eq*: *coord X v* = *coord X w*
shows *v* = *w*
proof −
have $\forall i.\ (THE\ f.\ \forall i.\ v\ \$\ i = (\sum x{\in}UNIV.\ f\ x\ *\ X\ \$\ x\ \$\ i))\ i = (THE\ f.\ \forall i.\ w\ \$\ i = (\sum x{\in}UNIV.\ f\ x\ *\ X\ \$\ x\ \$\ i))\ i$ **using** *coord-eq*

unfolding *coord-eq coord-def vec-eq-iff* **by** *simp*
hence *the-eq*: $(THE\ f.\ \forall i.\ v\ \$\ i = (\sum x{\in}UNIV.\ f\ x * X\ \$\ x\ \$\ i)) = (THE\ f.\ \forall i.\ w\ \$\ i = (\sum x{\in}UNIV.\ f\ x * X\ \$\ x\ \$\ i))$ **by** *auto*
obtain *f* **where** *f*: $(\sum x{\in}UNIV.\ f\ x\ *s\ X\ \$\ x) = v$ **using** *basis-UNIV*[*OF basis-X*] **by** *blast*
obtain *g* **where** *g*: $(\sum x{\in}UNIV.\ g\ x\ *s\ X\ \$\ x) = w$ **using** *basis-UNIV*[*OF basis-X*] **by** *blast*
have *the-f*: $(THE\ f.\ \forall i.\ v\ \$\ i = (\sum x{\in}UNIV.\ f\ x * X\ \$\ x\ \$\ i)) = f$
proof (*rule the-equality*)
show $\forall i.\ v\ \$\ i = (\sum x{\in}UNIV.\ f\ x * X\ \$\ x\ \$\ i)$ **using** *f* **by** *auto*
fix *fa* **assume** $\forall i.\ v\ \$\ i = (\sum x{\in}UNIV.\ fa\ x * X\ \$\ x\ \$\ i)$
hence $\forall i.\ v\ \$\ i = (\sum x{\in}UNIV.\ fa\ x\ *s\ X\ \$\ x)\ \$\ i$ **unfolding** *sum-component* **by** *simp*
hence *fa*: $v = (\sum x{\in}UNIV.\ fa\ x\ *s\ X\ \$\ x)$ **unfolding** *vec-eq-iff* **.**
show $fa = f$ **using** *basis-combination-unique*[*OF basis-X*] *f fa* **by** *simp*
qed
have *the-g*: $(THE\ g.\ \forall i.\ w\ \$\ i = (\sum x{\in}UNIV.\ g\ x * X\ \$\ x\ \$\ i)) = g$
proof (*rule the-equality*)
show $\forall i.\ w\ \$\ i = (\sum x{\in}UNIV.\ g\ x * X\ \$\ x\ \$\ i)$ **using** *g* **by** *auto*
fix *fa* **assume** $\forall i.\ w\ \$\ i = (\sum x{\in}UNIV.\ fa\ x * X\ \$\ x\ \$\ i)$
hence $\forall i.\ w\ \$\ i = (\sum x{\in}UNIV.\ fa\ x\ *s\ X\ \$\ x)\ \$\ i$ **unfolding** *sum-component* **by** *simp*
hence *fa*: $w = (\sum x{\in}UNIV.\ fa\ x\ *s\ X\ \$\ x)$ **unfolding** *vec-eq-iff* **.**
show $fa = g$ **using** *basis-combination-unique*[*OF basis-X*] *g fa* **by** *simp*
qed
have *f*=*g* **using** *the-eq* **unfolding** *the-f the-g* **.**
thus *v*=*w* **using** *f g* **by** *blast*
qed

7.5 Matrix of change of basis and coordinate matrix of a linear map

Definitions of matrix of change of basis and matrix of a linear transformation with respect to two bases:

definition *matrix-change-of-basis* :: $'a{::}\{field\}^{\frown}{}'n^{\frown}{}'n \Rightarrow {}'a^{\frown}{}'n^{\frown}{}'n \Rightarrow {}'a^{\frown}{}'n^{\frown}{}'n$
where *matrix-change-of-basis X Y* = $(\chi\ i\ j.\ (coord\ Y\ (X\$j))\ \$\ i)$

There exists in the library the definition $matrix\ ?f = (\chi i\ j.\ ?f\ (axis\ j\ 1)\ \$\ i)$, which is the coordinate matrix of a linear map with respect to the standard bases. Now we generalise that concept to the coordinate matrix of a linear map with respect to any two bases.

definition $matrix' :: 'a{::}\{field\}^{\frown}{}'n^{\frown}{}'n \Rightarrow {}'a^{\frown}{}'m^{\frown}{}'m \Rightarrow ({}'a^{\frown}{}'n => {}'a^{\frown}{}'m) \Rightarrow {}'a^{\frown}{}'n^{\frown}{}'m$
where *matrix' X Y f*= $(\chi\ i\ j.\ (coord\ Y\ (f(X\$j)))\ \$\ i)$

Properties of $matrix'\ ?X\ ?Y\ ?f = (\chi i\ j.\ coord\ ?Y\ (?f\ (?X\ \$\ j))\ \$\ i)$

lemma *matrix'-eq-matrix*:
defines *cart-basis-Rn*: *cart-basis-Rn* == $(cart\text{-}basis'){::}'a{::}\{field\}^{\frown}{}'n^{\frown}{}'n$
and *cart-basis-Rm*:*cart-basis-Rm* == $(cart\text{-}basis'){::}'a^{\frown}{}'m^{\frown}{}'m$

shows *matrix′* (*cart-basis-Rn*) (*cart-basis-Rm*) *f* = *matrix f*
proof (*unfold matrix-def matrix′-def coord-def*, *vector*, *auto*)
fix *i j*
have *basis-Rn*:*is-basis* (*set-of-vector cart-basis-Rn*) **using** *is-basis-cart-basis′* **unfolding** *cart-basis-Rn* **.**
have *basis-Rm*:*is-basis* (*set-of-vector cart-basis-Rm*) **using** *is-basis-cart-basis′* **unfolding** *cart-basis-Rm* **.**
obtain *g* **where** *sum-g*: (∑ *x*∈*UNIV*. *g x* ∗*s* (*cart-basis-Rm* $ *x*)) = *f* (*cart-basis-Rn* $ *j*) **using** *basis-UNIV*[*OF basis-Rm*] **by** *blast*
have *the-g*: (*THE g*. ∀ *a*. *f* (*cart-basis-Rn* $ *j*) $ *a* = (∑ *x*∈*UNIV*. *g x* ∗ *cart-basis-Rm* $ *x* $ *a*)) = *g*
proof (*rule the-equality*, *clarify*)
fix *a*
have *f* (*cart-basis-Rn* $ *j*) $ *a* = (∑ *i*∈*UNIV*. *g i* ∗*s* (*cart-basis-Rm* $ *i*)) $ *a* **using** *sum-g* **by** *simp*
also have ... = (∑ *x*∈*UNIV*. *g x* ∗ *cart-basis-Rm* $ *x* $ *a*) **unfolding** *sum-component* **by** *simp*
finally show *f* (*cart-basis-Rn* $ *j*) $ *a* = (∑ *x*∈*UNIV*. *g x* ∗ *cart-basis-Rm* $ *x* $ *a*) **.**
fix *ga* **assume** ∀ *a*. *f* (*cart-basis-Rn* $ *j*) $ *a* = (∑ *x*∈*UNIV*. *ga x* ∗ *cart-basis-Rm* $ *x* $ *a*)
hence *sum-ga*: *f* (*cart-basis-Rn* $ *j*) = (∑ *i*∈*UNIV*. *ga i* ∗*s cart-basis-Rm* $ *i*) **by** (*vector*, *auto*)
show *ga* = *g*
proof (*rule basis-combination-unique*)
show *is-basis* (*set-of-vector* (*cart-basis-Rm*)) **using** *basis-Rm* **.**
show (∑ *i*∈*UNIV*. *g i* ∗*s cart-basis-Rm* $ *i*) = (∑ *i*∈*UNIV*. *ga i* ∗*s cart-basis-Rm* $ *i*) **using** *sum-g sum-ga* **by** *simp*
qed
qed
show (*THE fa*. ∀ *i*. *f* (*cart-basis-Rn* $ *j*) $ *i* = (∑ *x*∈*UNIV*. *fa x* ∗ *cart-basis-Rm* $ *x* $ *i*)) *i* = *f* (*axis j 1*) $ *i*
unfolding *the-g* **using** *sum-g* **unfolding** *cart-basis-Rm cart-basis-Rn cart-basis′-def* **using** *basis-expansion-unique*[*of g f* (*axis j 1*)]
unfolding *scalar-mult-eq-scaleR* **by** *auto*
qed

lemma *matrix′*:
assumes *basis-X*: *is-basis* (*set-of-vector X*) **and** *basis-Y*: *is-basis* (*set-of-vector Y*)
shows *f* (*X*$*i*) = *sum* (λ*j*. (*matrix′ X Y f*) $ *j* $ *i* ∗*s* (*Y*$*j*)) *UNIV*
proof (*unfold matrix′-def coord-def matrix-mult-sum column-def*, *vector*, *auto*)
fix *j*
obtain *g* **where** *g*: (∑ *x*∈*UNIV*. *g x* ∗*s Y* $ *x*) = *f* (*X* $ *i*) **using** *basis-UNIV*[*OF basis-Y*] **by** *blast*
have *the-g*: (*THE fa*. ∀ *ia*. *f* (*X* $ *i*) $ *ia* = (∑ *x*∈*UNIV*. *fa x* ∗ *Y* $ *x* $ *ia*)) = *g*
proof (*rule the-equality*, *clarify*)
fix *a*

have $f\ (X\ \$\ i)\ \$\ a = (\sum x{\in}UNIV.\ g\ x\ {*s}\ Y\ \$\ x)\ \$\ a$ **using** g **by** *simp*
also have $... = (\sum x{\in}UNIV.\ g\ x\ *\ Y\ \$\ x\ \$\ a)$ **unfolding** *sum-component* **by** *auto*
finally show $f\ (X\ \$\ i)\ \$\ a = (\sum x{\in}UNIV.\ g\ x\ *\ Y\ \$\ x\ \$\ a)$ **.**
fix *fa*
assume $\forall ia.\ f\ (X\ \$\ i)\ \$\ ia = (\sum x{\in}UNIV.\ fa\ x\ *\ Y\ \$\ x\ \$\ ia)$
hence $\forall ia.\ f\ (X\ \$\ i)\ \$\ ia = (\sum x{\in}UNIV.\ fa\ x\ {*s}\ Y\ \$\ x)\ \$\ ia$ **unfolding** *sum-component* **by** *simp*
hence *fa*:$f\ (X\ \$\ i) = (\sum x{\in}UNIV.\ fa\ x\ {*s}\ Y\ \$\ x)$ **unfolding** *vec-eq-iff* **.**
show $fa = g$ **by** (*rule basis-combination-unique*[*OF basis-Y*], *simp add*: *fa g*)
qed
show $f\ (X\ \$\ i)\ \$\ j = (\sum x{\in}UNIV.\ (THE\ fa.\ \forall j.\ f\ (X\ \$\ i)\ \$\ j = (\sum x{\in}UNIV.\ fa\ x\ *\ Y\ \$\ x\ \$\ j))\ x\ *\ Y\ \$\ x\ \$\ j)$
unfolding *the-g* **unfolding** *g*[*symmetric*] *sum-component* **by** *simp*
qed

corollary *matrix'2*:
assumes *basis-X*: *is-basis* (*set-of-vector X*) **and** *basis-Y*: *is-basis* (*set-of-vector Y*)
and *eq-f*: $\forall i.\ f\ (X\$i) = sum\ (\lambda j.\ A\ \$\ j\ \$\ i\ {*s}\ (Y\$j))\ UNIV$
shows $matrix'\ X\ Y\ f = A$
proof −
have *eq-f'*: $\forall i.\ f\ (X\$i) = sum\ (\lambda j.\ (matrix'\ X\ Y\ f)\ \$\ j\ \$\ i\ {*s}\ (Y\$j))\ UNIV$ **using** *matrix'*[*OF basis-X basis-Y*] **by** *auto*
show *?thesis*
proof (*vector*, *auto*)
fix *i j*
define *a* **where** $a\ x = (matrix'\ X\ Y\ f)\ \$\ x\ \$\ i$ **for** *x*
define *b* **where** $b\ x = A\ \$\ x\ \$\ i$ **for** *x*
have *fxi-1*:$f\ (X\$i) = sum\ (\lambda j.\ a\ j\ {*s}\ (Y\$j))\ UNIV$ **using** *eq-f'* **unfolding** *a-def* **by** *simp*
have *fxi-2*: $f\ (X\$i) = sum\ (\lambda j.\ b\ j\ {*s}(Y\$j))\ UNIV$ **using** *eq-f* **unfolding** *b-def* **by** *simp*
have *a*=*b* **using** *basis-combination-unique*[*OF basis-Y*] *fxi-1 fxi-2* **by** *auto*
thus $(matrix'\ X\ Y\ f)\ \$\ j\ \$\ i = A\ \$\ j\ \$\ i$ **unfolding** *a-def b-def* **by** *metis*
qed
qed

This is the theorem 2.14 in the book "Advanced Linear Algebra" by Steven Roman.

lemma *coord-matrix'*:
fixes $X::'a::\{field\}\hat{}'n\hat{}'n$ **and** $Y::'a\hat{}'m\hat{}'m$
assumes *basis-X*: *is-basis* (*set-of-vector X*) **and** *basis-Y*: *is-basis* (*set-of-vector Y*)
and *linear-f*: *linear* $((*s))\ ((*s))\ f$
shows $coord\ Y\ (f\ v) = (matrix'\ X\ Y\ f)\ *v\ (coord\ X\ v)$
proof (*unfold matrix-mult-sum matrix'-def column-def coord-def*, *vector*, *auto*)
fix *i*

interpret *Vector-Spaces.linear* $(*s)$ $(*s)$ f **by** *fact*
obtain g **where** g: $(\sum x{\in}UNIV.\ g\ x\ *s\ Y\ \$\ x) = f\ v$ **using** *basis-UNIV*[*OF basis-Y*] **by** *auto*
obtain s **where** s: $(\sum x{\in}UNIV.\ s\ x\ *s\ X\ \$\ x) = v$ **using** *basis-UNIV*[*OF basis-X*] **by** *auto*
have *the-g*: $(THE\ fa.\ \forall a.\ f\ v\ \$\ a = (\sum x{\in}UNIV.\ fa\ x\ *\ Y\ \$\ x\ \$\ a)) = g$
proof (*rule the-equality*)
have $\forall a.\ f\ v\ \$\ a = (\sum x{\in}UNIV.\ g\ x\ *s\ Y\ \$\ x)\ \$\ a$ **using** g **by** *simp*
thus $\forall a.\ f\ v\ \$\ a = (\sum x{\in}UNIV.\ g\ x\ *\ Y\ \$\ x\ \$\ a)$ **unfolding** *sum-component* **by** *simp*
fix *fa* **assume** $\forall a.\ f\ v\ \$\ a = (\sum x{\in}UNIV.\ fa\ x\ *\ Y\ \$\ x\ \$\ a)$
hence *fa*: $f\ v = (\sum x{\in}UNIV.\ fa\ x\ *s\ Y\ \$\ x)$ **by** (*vector*, *auto*)
show *fa*=g **by** (*rule basis-combination-unique*[*OF basis-Y*], *simp add*: *fa g*)
qed
have *the-s*: $(THE\ f.\ \forall i.\ v\ \$\ i = (\sum x{\in}UNIV.\ f\ x\ *\ X\ \$\ x\ \$\ i)){=}s$
proof (*rule the-equality*)
have $\forall i.\ v\ \$\ i = (\sum x{\in}UNIV.\ s\ x\ *s\ X\ \$\ x)\ \$\ i$ **using** s **by** *simp*
thus $\forall i.\ v\ \$\ i = (\sum x{\in}UNIV.\ s\ x\ *\ X\ \$\ x\ \$\ i)$**unfolding** *sum-component* **by** *simp*
fix *fa* **assume** $\forall i.\ v\ \$\ i = (\sum x{\in}UNIV.\ fa\ x\ *\ X\ \$\ x\ \$\ i)$
hence *fa*: $v{=}(\sum x{\in}UNIV.\ fa\ x\ *s\ X\ \$\ x)$ **by** (*vector*, *auto*)
show *fa*=s**by** (*rule basis-combination-unique*[*OF basis-X*], *simp add*: *fa s*)
qed
define t **where** $t\ x = (\sum i{\in}UNIV.\ (s\ i\ *\ (THE\ fa.\ f\ (X\ \$\ i) = (\sum x{\in}UNIV.\ fa\ x\ *s\ Y\ \$\ x))\ x))$ **for** x
have $(\sum x{\in}UNIV.\ g\ x\ *s\ Y\ \$\ x) = f\ v$ **using** g **by** *simp*
also have $... = f\ (\sum x{\in}UNIV.\ s\ x\ *s\ X\ \$\ x)$ **using** s **by** *simp*
also have $... = (\sum x{\in}UNIV.\ s\ x\ *s\ f\ (X\ \$\ x))$ **by** (*simp add*: *sum scale*)
also have $... = (\sum i{\in}UNIV.\ s\ i\ *s\ sum\ (\lambda j.\ (matrix'\ X\ Y\ f)\$j\$i\ *s\ (Y\$j))\ UNIV)$ **using** *matrix'*[*OF basis-X basis-Y*] **by** *auto*
also have $... = (\sum i{\in}UNIV.\ \sum x{\in}UNIV.\ s\ i\ *s\ (matrix'\ X\ Y\ f\ \$\ x\ \$\ i\ *s\ Y\ \$\ x))$ **unfolding** *vec.scale-sum-right* **..**
also have $... = (\sum i{\in}UNIV.\ \sum x{\in}UNIV.\ (s\ i\ *\ (THE\ fa.\ f\ (X\ \$\ i) = (\sum x{\in}UNIV.\ fa\ x\ *s\ Y\ \$\ x))\ x)\ *s\ Y\ \$\ x)$ **unfolding** *matrix'-def* **unfolding** *coord-def* **by** *auto*
also have $... = (\sum x{\in}UNIV.\ (\sum i{\in}UNIV.\ (s\ i\ *\ (THE\ fa.\ f\ (X\ \$\ i) = (\sum x{\in}UNIV.\ fa\ x\ *s\ Y\ \$\ x))\ x)\ *s\ Y\ \$\ x))$
by (*rule sum.swap*)
also have $... = (\sum x{\in}UNIV.\ (\sum i{\in}UNIV.\ (s\ i\ *\ (THE\ fa.\ f\ (X\ \$\ i) = (\sum x{\in}UNIV.\ fa\ x\ *s\ Y\ \$\ x))\ x))\ *s\ Y\ \$\ x)$ **unfolding** *vec.scale-sum-left* **..**
also have $... = (\sum x{\in}UNIV.\ t\ x\ *s\ Y\ \$\ x)$ **unfolding** *t-def* **..**
finally have $(\sum x{\in}UNIV.\ g\ x\ *s\ Y\ \$\ x) = (\sum x{\in}UNIV.\ t\ x\ *s\ Y\ \$\ x)$ **.**
hence $g{=}t$ **using** *basis-combination-unique*[*OF basis-Y*] **by** *simp*
thus $(THE\ fa.\ \forall i.\ f\ v\ \$\ i = (\sum x{\in}UNIV.\ fa\ x\ *\ Y\ \$\ x\ \$\ i))\ i =$
$(\sum x{\in}UNIV.\ (THE\ f.\ \forall i.\ v\ \$\ i = (\sum x{\in}UNIV.\ f\ x\ *\ X\ \$\ x\ \$\ i))\ x\ *\ (THE\ fa.\ \forall i.\ f\ (X\ \$\ x)\ \$\ i = (\sum x{\in}UNIV.\ fa\ x\ *\ Y\ \$\ x\ \$\ i))\ i)$
proof (*unfold the-g the-s t-def*, *auto*)
have $(\sum x{\in}UNIV.\ s\ x\ *\ (THE\ fa.\ \forall i.\ f\ (X\ \$\ x)\ \$\ i = (\sum x{\in}UNIV.\ fa\ x\ *\ Y\ \$\ x\ \$\ i))\ i) =$
$(\sum x{\in}UNIV.\ s\ x\ *\ (THE\ fa.\ \forall i.\ f\ (X\ \$\ x)\ \$\ i = (\sum x{\in}UNIV.\ fa\ x\ *s\ Y\ \$$

x) \$ i) i) **unfolding** *sum-component* **by** *simp*
also have ... = ($\sum x{\in}UNIV.\ s\ x * (THE\ fa.\ f\ (X\ \$\ x) = (\sum x{\in}UNIV.\ fa\ x *s\ Y\ \$\ x))\ i)$ **by** (*rule sum.cong*, *auto simp add*: *vec-eq-iff*)
finally show ($\sum ia{\in}UNIV.\ s\ ia * (THE\ fa.\ f\ (X\ \$\ ia) = (\sum x{\in}UNIV.\ fa\ x *s\ Y\ \$\ x))\ i) = (\sum x{\in}UNIV.\ s\ x * (THE\ fa.\ \forall i.\ f\ (X\ \$\ x)\ \$\ i = (\sum x{\in}UNIV.\ fa\ x * Y\ \$\ x\ \$\ i))\ i)$
by *auto*
qed
qed

This is the second part of the theorem 2.15 in the book "Advanced Linear Algebra" by Steven Roman.

lemma *matrix'-compose*:
fixes $X::'a::\{field\}\hat{}'n\hat{}'n$ **and** $Y::'a\hat{}'m\hat{}'m$ **and** $Z::'a\hat{}'p\hat{}'p$
assumes *basis-X*: *is-basis* (*set-of-vector* X) **and** *basis-Y*: *is-basis* (*set-of-vector* Y) **and** *basis-Z*: *is-basis* (*set-of-vector* Z)
and *linear-f*: *linear* ((∗*s*)) ((∗*s*)) f **and** *linear-g*: *linear* ((∗*s*)) ((∗*s*)) g
shows *matrix'* $X\ Z\ (g \circ f) = (matrix'\ Y\ Z\ g) **\ (matrix'\ X\ Y\ f)$
proof (*unfold matrix-eq*, *clarify*)
fix $a::('a, 'n)\ vec$
obtain v **where** v: $a = coord\ X\ v$ **using** *bij-coord*[*OF basis-X*]
by (*meson bij-pointE*)
have *linear-gf*: *linear* ((∗*s*)) ((∗*s*)) $(g \circ f)$
using *Vector-Spaces.linear-compose*[*OF linear-f linear-g*] **.**
have $matrix'\ X\ Z\ (g \circ f) *v\ a = matrix'\ X\ Z\ (g \circ f) *v\ (coord\ X\ v)$ **unfolding** v **..**
also have ... = $coord\ Z\ ((g \circ f)\ v)$ **unfolding** *coord-matrix'*[*OF basis-X basis-Z linear-gf*, *symmetric*] **..**
also have ... = $coord\ Z\ (g\ (f\ v))$ **unfolding** *o-def* **..**
also have ... = $(matrix'\ Y\ Z\ g) *v\ (coord\ Y\ (f\ v))$ **unfolding** *coord-matrix'*[*OF basis-Y basis-Z linear-g*] **..**
also have ... = $(matrix'\ Y\ Z\ g) *v\ ((matrix'\ X\ Y\ f) *v\ (coord\ X\ v))$ **unfolding** *coord-matrix'*[*OF basis-X basis-Y linear-f*] **..**
also have ... = $((matrix'\ Y\ Z\ g) **\ (matrix'\ X\ Y\ f)) *v\ (coord\ X\ v)$ **unfolding** *matrix-vector-mul-assoc* **..**
finally show $matrix'\ X\ Z\ (g \circ f) *v\ a = matrix'\ Y\ Z\ g **\ matrix'\ X\ Y\ f *v\ a$ **unfolding** v **.**
qed

lemma *exists-linear-eq-matrix'*:
fixes $A::'a::\{field\}\hat{}'m\hat{}'n$ **and** $X::'a\hat{}'m\hat{}'m$ **and** $Y::'a\hat{}'n\hat{}'n$
assumes *basis-X*: *is-basis* (*set-of-vector* X) **and** *basis-Y*: *is-basis* (*set-of-vector* Y)
shows $\exists f.\ matrix'\ X\ Y\ f = A \wedge linear\ ((*s))\ ((*s))\ f$
proof −
define f **where** $f\ v = sum\ (\lambda j.\ A\ \$\ j\ \$\ (THE\ k.\ v = X\ \$\ k) *s\ Y\ \$\ j)\ UNIV$ **for** v
have *indep*: *vec.independent* (*set-of-vector* X)

using *basis-X* **unfolding** *is-basis-def* **by** *auto*
define *g* **where** $g = vec.construct\ (set\text{-}of\text{-}vector\ X)\ f$
have *linear-g*: $linear\ ((*s))\ ((*s))\ g$ **and** *f-eq-g*: $(\forall x \in (set\text{-}of\text{-}vector\ X).\ g\ x = f\ x)$
using *vec.linear-construct*[*OF indep*] *vec.construct-basis*[*OF indep*]
unfolding *g-def module-hom-iff-linear*
by *auto*
show *?thesis*
proof (*rule exI*[*of - g*], *rule conjI*)
show $matrix'\ X\ Y\ g = A$
proof (*rule matrix'2*)
show $is\text{-}basis\ (set\text{-}of\text{-}vector\ X)$ **using** *basis-X* **.**
show $is\text{-}basis\ (set\text{-}of\text{-}vector\ Y)$ **using** *basis-Y* **.**
show $\forall i.\ g\ (X\ \$\ i) = (\sum j{\in}UNIV.\ A\ \$\ j\ \$\ i *s\ Y\ \$\ j)$
proof (*clarify*)
fix *i*
have *the-k-eq-i*: $(THE\ k.\ X\ \$\ i = X\ \$\ k) = i$
proof (*rule the-equality*)
show $X\ \$\ i = X\ \$\ i$ **..**
fix *k* **assume** *Xi-Xk*: $X\ \$\ i = X\ \$\ k$ **show** $k = i$ **using** *Xi-Xk basis-X inj-eq inj-op-nth* **by** *metis*
qed
have *Xi-in-X*:$X\$i \in (set\text{-}of\text{-}vector\ X)$ **unfolding** *set-of-vector-def* **by** *auto*
have $g\ (X\$i) = f\ (X\$i)$ **using** *f-eq-g Xi-in-X* **by** *simp*
also have $... = (\sum j{\in}UNIV.\ A\ \$\ j\ \$\ (THE\ k.\ X\ \$\ i = X\ \$\ k) *s\ Y\ \$\ j)$ **unfolding** *f-def* **..**
also have $... = (\sum j{\in}UNIV.\ A\ \$\ j\ \$\ i *s\ Y\ \$\ j)$ **unfolding** *the-k-eq-i* **..**
finally show $g\ (X\ \$\ i) = (\sum j{\in}UNIV.\ A\ \$\ j\ \$\ i *s\ \ Y\ \$\ j)$ **.**
qed
qed
show $linear\ ((*s))\ ((*s))\ g$ **using** *linear-g* **.**
qed
qed

lemma *matrix'-surj*:
assumes *basis-X*: $is\text{-}basis\ (set\text{-}of\text{-}vector\ X)$ **and** *basis-Y*: $is\text{-}basis\ (set\text{-}of\text{-}vector\ Y)$
shows $surj\ (matrix'\ X\ Y)$
proof (*unfold surj-def*, *clarify*)
fix *A*
show $\exists f.\ A = matrix'\ X\ Y\ f$
using *exists-linear-eq-matrix'*[*OF basis-X basis-Y*, *of A*] **unfolding** *matrix'-def* **by** *auto*
qed

Properties of $matrix\text{-}change\text{-}of\text{-}basis\ ?X\ ?Y = (\chi i\ j.\ coord\ ?Y\ (?X\ \$\ j)\ \$\ i)$.

This is the first part of the theorem 2.12 in the book "Advanced Linear

Algebra" by Steven Roman.

lemma *matrix-change-of-basis-works*:
fixes $X::'a::\{field\}\hat{}'n\hat{}'n$ **and** $Y::'a\hat{}'n\hat{}'n$
assumes *basis-X*: *is-basis* (*set-of-vector X*)
and *basis-Y*: *is-basis* (*set-of-vector Y*)
shows (*matrix-change-of-basis X Y*) $*v$ (*coord X v*) = (*coord Y v*)
proof (*unfold matrix-mult-sum matrix-change-of-basis-def column-def coord-def*, *vector*, *auto*)
fix *i*
obtain *f* **where** *f*: $(\sum x{\in}UNIV.\ f\ x\ *s\ Y\ \$\ x) = v$ **using** *basis-UNIV*[*OF basis-Y*] **by** *blast*
obtain *g* **where** *g*: $(\sum x{\in}UNIV.\ g\ x\ *s\ X\ \$\ x) = v$ **using** *basis-UNIV*[*OF basis-X*] **by** *blast*
define *t* **where** $t\ x = (THE\ f.\ X\ \$\ x{=}\ (\sum a{\in}UNIV.\ f\ a\ *s\ Y\ \$\ a))$ **for** *x*
define *w* **where** $w\ i = (\sum x{\in}UNIV.\ g\ x * t\ x\ i)$ **for** *i*
have *the-f*:$(THE\ f.\ \forall i.\ v\ \$\ i = (\sum x{\in}UNIV.\ f\ x * Y\ \$\ x\ \$\ i)) = f$
proof (*rule the-equality*)
show $\forall i.\ v\ \$\ i = (\sum x{\in}UNIV.\ f\ x * Y\ \$\ x\ \$\ i)$ **using** *f* **by** *auto*
fix *fa* **assume** $\forall i.\ v\ \$\ i = (\sum x{\in}UNIV.\ fa\ x * Y\ \$\ x\ \$\ i)$
hence $\forall i.\ v\ \$\ i = (\sum x{\in}UNIV.\ fa\ x\ *s\ Y\ \$\ x)\ \$\ i$ **unfolding** *sum-component* **by** *simp*
hence *fa*: $v = (\sum x{\in}UNIV.\ fa\ x\ *s\ Y\ \$\ x)$ **unfolding** *vec-eq-iff* **.**
show $fa = f$
using *basis-combination-unique*[*OF basis-Y*] *fa f* **by** *simp*
qed
have *the-g*: $(THE\ f.\ \forall i.\ v\ \$\ i = (\sum x{\in}UNIV.\ f\ x * X\ \$\ x\ \$\ i)) = g$
proof (*rule the-equality*)
show $\forall i.\ v\ \$\ i = (\sum x{\in}UNIV.\ g\ x * X\ \$\ x\ \$\ i)$ **using** *g* **by** *auto*
fix *fa* **assume** $\forall i.\ v\ \$\ i = (\sum x{\in}UNIV.\ fa\ x * X\ \$\ x\ \$\ i)$
hence $\forall i.\ v\ \$\ i = (\sum x{\in}UNIV.\ fa\ x\ *s\ X\ \$\ x)\ \$\ i$ **unfolding** *sum-component* **by** *simp*
hence *fa*: $v = (\sum x{\in}UNIV.\ fa\ x\ *s\ X\ \$\ x)$ **unfolding** *vec-eq-iff* **.**
show $fa = g$
using *basis-combination-unique*[*OF basis-X*] *fa g* **by** *simp*
qed
have $(\sum x{\in}UNIV.\ f\ x\ *s\ Y\ \$\ x) = (\sum x{\in}UNIV.\ g\ x\ *s\ X\ \$\ x)$ **unfolding** *f g* **..**
also have ... = $(\sum x{\in}UNIV.\ g\ x\ *s\ (sum\ \ (\lambda i.\ (t\ x\ i)\ *s\ Y\ \$\ i)\ \ UNIV))$ **unfolding** *t-def*
proof (*rule sum.cong*)
fix *x*
obtain *h* **where** *h*: $(\sum a{\in}UNIV.\ h\ a\ *s\ Y\ \$\ a) = X\$x$ **using** *basis-UNIV*[*OF basis-Y*] **by** *blast*
have *the-h*: $(THE\ f.\ X\ \$\ x = (\sum a{\in}UNIV.\ f\ a\ *s\ Y\ \$\ a)){=}h$
proof (*rule the-equality*)
show $X\ \$\ x = (\sum a{\in}UNIV.\ h\ a\ *s\ Y\ \$\ a)$ **using** *h* **by** *simp*
fix *f* **assume** *f*: $X\ \$\ x = (\sum a{\in}UNIV.\ f\ a\ *s\ Y\ \$\ a)$
show $f = h$ **using** *basis-combination-unique*[*OF basis-Y*] *f h* **by** *simp*
qed
show $g\ x\ *s\ X\ \$\ x = g\ x\ *s\ (\sum i{\in}UNIV.\ (THE\ f.\ X\ \$\ x = (\sum a{\in}UNIV.\ f\ a$

$*s$ Y \$ a)) i $*s$ Y \$ i) **unfolding** *the-h h* **..**
qed *rule*
also have ... = ($\sum x{\in}UNIV$. (*sum* (λi. $g\ x\ *s$ (($t\ x\ i$) $*s$ Y \$ i)) $UNIV$)) **unfolding** *vec.scale-sum-right* **..**
also have ... = ($\sum i{\in}UNIV$. $\sum x{\in}UNIV$. $g\ x\ *s$ ($t\ x\ i\ *s$ Y \$ i)) **by** (*rule sum.swap*)
also have ... = ($\sum i{\in}UNIV$. ($\sum x{\in}UNIV$. $g\ x * t\ x\ i$) $*s$ Y \$ i) **unfolding** *vec.scale-sum-left* **by** *auto*
finally have ($\sum x{\in}UNIV$. $f\ x\ *s$ Y \$ x) = ($\sum i{\in}UNIV$. ($\sum x{\in}UNIV$. $g\ x * t\ x\ i$) $*s$ Y \$ i) **.**
hence $f{=}w$ **using** *basis-combination-unique*[*OF basis-Y*] **unfolding** *w-def* **by** *auto*
thus ($\sum x{\in}UNIV$. (*THE f*. $\forall i$. v \$ i = ($\sum x{\in}UNIV$. $f\ x * X$ \$ x \$ i)) x * (*THE f*. $\forall i$. X \$ x \$ i = ($\sum x{\in}UNIV$. $f\ x *$ Y \$ x \$ i)) i) =
(*THE f*. $\forall i$. v \$ i = ($\sum x{\in}UNIV$. $f\ x *$ Y \$ x \$ i)) i **unfolding** *the-f the-g* **unfolding** *w-def t-def* **unfolding** *vec-eq-iff* **by** *auto*
qed

lemma *matrix-change-of-basis-mat-1*:
fixes X::$'a$::{*field*}$^{\frown}{}'n^{\frown}{}'n$
assumes *basis-X*: *is-basis* (*set-of-vector X*)
shows *matrix-change-of-basis X X* = *mat 1*
proof (*unfold matrix-change-of-basis-def coord-def mat-def*, *vector*, *auto*)
fix j::$'n$
define f :: $'n \Rightarrow 'a$ **where** $f\ i$ = (*if i=j then 1 else 0*) **for** i
have *UNIV-rw*: *UNIV* = *insert j* (*UNIV*−{j}) **by** *auto*
have ($\sum x{\in}UNIV$. $f\ x\ *s$ X \$ x) = ($\sum x{\in}$(*insert j* (*UNIV*−{j})). $f\ x\ *s$ X \$ x) **using** *UNIV-rw* **by** *simp*
also have ... = (λx. $f\ x\ *s$ X \$ x) j + ($\sum x{\in}$(*UNIV*−{j}). $f\ x\ *s$ X \$ x) **by** (*rule sum.insert*, *simp+*)
also have ... = X\$$j$ + ($\sum x{\in}$(*UNIV*−{j}). $f\ x\ *s$ X \$ x) **unfolding** *f-def* **by** *simp*
also have ... = X\$$j$ + *0* **unfolding** *add-left-cancel f-def* **by** (*rule sum.neutral*, *simp*)
finally have f: ($\sum x{\in}UNIV$. $f\ x\ *s$ X \$ x) = X\$$j$ **by** *simp*
have *the-f*: (*THE f*. $\forall i$. X \$ j \$ i = ($\sum x{\in}UNIV$. $f\ x * X$ \$ x \$ i)) = f
proof (*rule the-equality*)
show $\forall i$. X \$ j \$ i = ($\sum x{\in}UNIV$. $f\ x * X$ \$ x \$ i) **using** f **unfolding** *vec-eq-iff* **unfolding** *sum-component* **by** *simp*
fix *fa* **assume** $\forall i$. X \$ j \$ i = ($\sum x{\in}UNIV$. $fa\ x * X$ \$ x \$ i)
hence $\forall i$. X \$ j \$ i = ($\sum x{\in}UNIV$. $fa\ x\ *s\ X$ \$ x) \$ i **unfolding** *sum-component* **by** *simp*
hence *fa*: X \$ j = ($\sum x{\in}UNIV$. $fa\ x\ *s$ X \$ x) **unfolding** *vec-eq-iff* **.**
show $fa = f$ **using** *basis-combination-unique*[*OF basis-X*] *fa f* **by** *simp*
qed
show (*THE f*. $\forall i$. X \$ j \$ i = ($\sum x{\in}UNIV$. $f\ x * X$ \$ x \$ i)) j = *1* **unfolding** *the-f f-def* **by** *simp*

fix i **assume** *i-not-j*: $i \neq j$
show $(THE\ f.\ \forall i.\ X\ \$\ j\ \$\ i = (\sum x{\in}UNIV.\ f\ x * X\ \$\ x\ \$\ i))\ i = 0$ **unfolding** *the-f f-def* **using** *i-not-j* **by** *simp*
qed

Relationships between $matrix'\ ?X\ ?Y\ ?f = (\chi i\ j.\ coord\ ?Y\ (?f\ (?X\ \$\ j))\ \$\ i)$ and $matrix\text{-}change\text{-}of\text{-}basis\ ?X\ ?Y = (\chi i\ j.\ coord\ ?Y\ (?X\ \$\ j)\ \$\ i)$. This is the theorem 2.16 in the book "Advanced Linear Algebra" by Steven Roman.

lemma *matrix'-matrix-change-of-basis*:
fixes $B{::}'a{::}\{field\}^{\wedge}'n^{\wedge}'n$ **and** $B'{::}'a^{\wedge}'n^{\wedge}'n$ **and** $C{::}'a^{\wedge}'m^{\wedge}'m$ **and** $C'{::}'a^{\wedge}'m^{\wedge}'m$
assumes *basis-B*: *is-basis* (*set-of-vector B*) **and** *basis-B'*: *is-basis* (*set-of-vector B'*)
and *basis-C*: *is-basis* (*set-of-vector C*) **and** *basis-C'*: *is-basis* (*set-of-vector C'*)
and *linear-f*: *linear* ((∗s)) ((∗s)) *f*
shows $matrix'\ B'\ C'\ f = matrix\text{-}change\text{-}of\text{-}basis\ C\ C' **\ matrix'\ B\ C\ f **\ matrix\text{-}change\text{-}of\text{-}basis\ B'\ B$
proof (*unfold matrix-eq*, *clarify*)
fix *x*
obtain *v* **where** *v*: *x*=*coord B' v* **using** *bij-coord*[*OF basis-B'*] **by** (*meson bij-pointE*)
have $matrix\text{-}change\text{-}of\text{-}basis\ C\ C' **\ matrix'\ B\ C\ f **\ matrix\text{-}change\text{-}of\text{-}basis\ B'\ B *v\ (coord\ B'\ v)$
$= matrix\text{-}change\text{-}of\text{-}basis\ C\ C' **\ matrix'\ B\ C\ f *v\ (matrix\text{-}change\text{-}of\text{-}basis\ B'\ B *v\ (coord\ B'\ v))$ **unfolding** *matrix-vector-mul-assoc* **..**
also have $... = matrix\text{-}change\text{-}of\text{-}basis\ C\ C' **\ matrix'\ B\ C\ f *v\ (coord\ B\ v)$ **unfolding** *matrix-change-of-basis-works*[*OF basis-B' basis-B*] **..**
also have $... = matrix\text{-}change\text{-}of\text{-}basis\ C\ C' *v\ (matrix'\ B\ C\ f *v\ (coord\ B\ v))$ **unfolding** *matrix-vector-mul-assoc* **..**
also have $... = matrix\text{-}change\text{-}of\text{-}basis\ C\ C' *v\ (coord\ C\ (f\ v))$ **unfolding** *coord-matrix'*[*OF basis-B basis-C linear-f*] **..**
also have $... = coord\ C'\ (f\ v)$ **unfolding** *matrix-change-of-basis-works*[*OF basis-C basis-C'*] **..**
also have $... = matrix'\ B'\ C'\ f *v\ coord\ B'\ v$ **unfolding** *coord-matrix'*[*OF basis-B' basis-C' linear-f*] **..**
finally show $matrix'\ B'\ C'\ f *v\ x = matrix\text{-}change\text{-}of\text{-}basis\ C\ C' **\ matrix'\ B\ C\ f **\ matrix\text{-}change\text{-}of\text{-}basis\ B'\ B *v\ x$ **unfolding** *v* **..**
qed

lemma *matrix'-id-eq-matrix-change-of-basis*:
fixes $X{::}'a{::}\{field\}^{\wedge}'n^{\wedge}'n$ **and** $Y{::}'a^{\wedge}'n^{\wedge}'n$
assumes *basis-X*: *is-basis* (*set-of-vector X*) **and** *basis-Y*: *is-basis* (*set-of-vector Y*)
shows $matrix'\ X\ Y\ (id) = matrix\text{-}change\text{-}of\text{-}basis\ X\ Y$
unfolding *matrix'-def matrix-change-of-basis-def* **unfolding** *id-def* **..**

Relationships among $invertible\text{-}lf\ ?s1.0\ ?s2.0\ ?f \equiv linear\ ?s1.0\ ?s2.0\ ?f \wedge invertible\text{-}lf\text{-}axioms\ ?f$, $matrix\text{-}change\text{-}of\text{-}basis\ ?X\ ?Y = (\chi i\ j.\ coord\ ?Y\ (?X\ \$\ j)\ \$\ i)$, $matrix'\ ?X\ ?Y\ ?f = (\chi i\ j.\ coord\ ?Y\ (?f\ (?X\ \$\ j))\ \$\ i)$ and $invertible\ ?A = (\exists A'.\ ?A **\ A' = mat\ 1 \wedge A' **\ ?A = mat\ 1)$.

This is the second part of the theorem 2.12 in the book "Advanced Linear Algebra" by Steven Roman.

lemma *matrix-inv-matrix-change-of-basis*:
fixes $X::'a::\{field\}\hat{}'n\hat{}'n$ **and** $Y::'a\hat{}'n\hat{}'n$
assumes *basis-X*: *is-basis* (*set-of-vector X*) **and** *basis-Y*: *is-basis* (*set-of-vector Y*)
shows*matrix-change-of-basis Y X* = *matrix-inv* (*matrix-change-of-basis X Y*)
proof (*rule matrix-inv-unique*[*symmetric*])
have *linear-id*: *linear* ((∗*s*)) ((∗*s*)) *id* **by** (*metis vec.linear-id*)
have (*matrix-change-of-basis Y X*) ∗∗ (*matrix-change-of-basis X Y*) = (*matrix' Y X id*) ∗∗ (*matrix' X Y id*)
unfolding *matrix'-id-eq-matrix-change-of-basis*[*OF basis-X basis-Y*]
unfolding *matrix'-id-eq-matrix-change-of-basis*[*OF basis-Y basis-X*] **..**
also have ... = *matrix' X X* (*id* ∘ *id*) **using** *matrix'-compose*[*OF basis-X basis-Y basis-X linear-id linear-id*] **..**
also have ... = *matrix-change-of-basis X X* **using** *matrix'-id-eq-matrix-change-of-basis*[*OF basis-X basis-X*] **unfolding** *o-def id-def* **.**
also have ... = *mat 1* **using** *matrix-change-of-basis-mat-1*[*OF basis-X*] **.**
finally show *matrix-change-of-basis Y X ∗∗ matrix-change-of-basis X Y* = *mat 1* **.**
have (*matrix-change-of-basis X Y*) ∗∗ (*matrix-change-of-basis Y X*) = (*matrix' X Y id*) ∗∗ (*matrix' Y X id*)
unfolding *matrix'-id-eq-matrix-change-of-basis*[*OF basis-X basis-Y*]
unfolding *matrix'-id-eq-matrix-change-of-basis*[*OF basis-Y basis-X*] **..**
also have ... = *matrix' Y Y* (*id* ∘ *id*) **using** *matrix'-compose*[*OF basis-Y basis-X basis-Y linear-id linear-id*] **..**
also have ... = *matrix-change-of-basis Y Y* **using** *matrix'-id-eq-matrix-change-of-basis*[*OF basis-Y basis-Y*] **unfolding** *o-def id-def* **.**
also have ... = *mat 1* **using** *matrix-change-of-basis-mat-1*[*OF basis-Y*] **.**
finally show *matrix-change-of-basis X Y ∗∗ matrix-change-of-basis Y X* = *mat 1* **.**
qed

The following four lemmas are the proof of the theorem 2.13 in the book "Advanced Linear Algebra" by Steven Roman.

corollary *invertible-matrix-change-of-basis*:
fixes $X::'a::\{field\}\hat{}'n\hat{}'n$ **and** $Y::'a\hat{}'n\hat{}'n$
assumes *basis-X*: *is-basis* (*set-of-vector X*) **and** *basis-Y*: *is-basis* (*set-of-vector Y*)
shows *invertible* (*matrix-change-of-basis X Y*)
by (*metis basis-X basis-Y invertible-left-inverse vec.linear-id*
matrix'-id-eq-matrix-change-of-basis matrix'-matrix-change-of-basis
matrix-change-of-basis-mat-1)

lemma *invertible-lf-imp-invertible-matrix'*:
fixes $f::'a::\{field\}\hat{}'b \Rightarrow 'a\hat{}'b$
assumes *invertible-lf* ((∗*s*)) ((∗*s*)) *f* **and** *basis-X*: *is-basis* (*set-of-vector X*) **and** *basis-Y*: *is-basis* (*set-of-vector Y*)
shows *invertible* (*matrix' X Y f*)

by (*metis* (*lifting*) *assms*(*1*) *basis-X basis-Y invertible-lf-def invertible-lf-imp-invertible-matrix invertible-matrix-change-of-basis invertible-mult is-basis-cart-basis' matrix'-eq-matrix matrix'-matrix-change-of-basis*)

lemma *invertible-matrix'-imp-invertible-lf*:
fixes *f*:: *'a*::{*field*}^'*b* ⇒ *'a*^'*b*
assumes *invertible* (*matrix' X Y f*) **and** *basis-X*: *is-basis* (*set-of-vector X*)
and *linear-f*: *linear* ((∗*s*)) ((∗*s*)) *f* **and** *basis-Y*: *is-basis* (*set-of-vector Y*)
shows *invertible-lf* ((∗*s*)) ((∗*s*)) *f*
unfolding *invertible-matrix-iff-invertible-lf'*[*OF linear-f*, *symmetric*]
by (*metis assms*(*1*) *basis-X basis-Y id-o invertible-matrix-change-of-basis invertible-mult is-basis-cart-basis' linear-f vec.linear-id matrix'-compose matrix'-eq-matrix matrix'-id-eq-matrix-change-of-basis matrix'-matrix-change-of-basis*)

lemma *invertible-matrix-is-change-of-basis*:
assumes *invertible-P*: *invertible P* **and** *basis-X*: *is-basis* (*set-of-vector X*)
shows ∃!*Y*. *matrix-change-of-basis Y X* = *P* ∧ *is-basis* (*set-of-vector Y*)
proof (*auto*)
show ∃ *Y*. *matrix-change-of-basis Y X* = *P* ∧ *is-basis* (*set-of-vector Y*)
proof −
fix *i j*
obtain *f* **where** *P*: *P* = *matrix' X X f* **and** *linear-f*: *linear* ((∗*s*)) ((∗*s*)) *f*
using *exists-linear-eq-matrix'*[*OF basis-X basis-X*, *of P*] **by** *blast*
show *?thesis*
proof (*rule exI*[*of* - χ *j*. *f* (*X*$*j*)], *rule conjI*)
show *matrix-change-of-basis* (χ *j*. *f* (*X* $ *j*)) *X* = *P* **unfolding** *matrix-change-of-basis-def P matrix'-def* **by** *vector*
have *invertible-f*: *invertible-lf* ((∗*s*)) ((∗*s*)) *f* **using** *invertible-matrix'-imp-invertible-lf*[*OF - basis-X linear-f basis-X*] **using** *invertible-P* **unfolding** *P* **by** *simp*
have *rw*: *set-of-vector* (χ *j*. *f* (*X* $ *j*)) = *f*‘(*set-of-vector X*) **unfolding** *set-of-vector-def* **by** *auto*
show *is-basis* (*set-of-vector* (χ *j*. *f* (*X* $ *j*))) **unfolding** *rw* **using** *basis-image-linear*[*OF invertible-f basis-X*] **.**
qed
qed
fix *Y Z*
assume *basis-Y*:*is-basis* (*set-of-vector Y*) **and** *eq*: *matrix-change-of-basis Z X* = *matrix-change-of-basis Y X* **and** *basis-Z*: *is-basis* (*set-of-vector Z*)
have *ZY-coord*: ∀ *i*. *coord X* (*Z*$*i*) = *coord X* (*Y*$*i*) **using** *eq* **unfolding** *matrix-change-of-basis-def* **unfolding** *vec-eq-iff* **by** *vector*
show *Y*=*Z* **by** (*vector*, *metis ZY-coord coord-eq*[*OF basis-X*])
qed

7.6 Equivalent Matrices

Next definition follows the one presented in Modern Algebra by Seth Warner.

definition *equivalent-matrices A B* = (∃ *P Q*. *invertible P* ∧ *invertible Q* ∧ *B* = (*matrix-inv P*)∗∗*A*∗∗*Q*)

lemma *exists-basis*: $\exists X::'a::\{field\}\hat{}'n\hat{}'n.$ *is-basis* (*set-of-vector* X)
using *is-basis-cart-basis'* **by** *auto*

lemma *equivalent-implies-exist-matrix'*:
assumes *equivalent*: *equivalent-matrices* A B
shows $\exists X\ Y\ X'\ Y'\ f::'a::\{field\}\hat{}'n \Rightarrow 'a\hat{}'m.$
linear $((*s))$ $((*s))$ $f \wedge$ *matrix'* $X\ Y\ f = A \wedge$ *matrix'* $X'\ Y'\ f = B \wedge$ *is-basis* (*set-of-vector* X)
$\wedge$ *is-basis* (*set-of-vector* Y) $\wedge$ *is-basis* (*set-of-vector* X') $\wedge$ *is-basis* (*set-of-vector* Y')
proof −
obtain $X::'a\hat{}'n\hat{}'n$ **where** X: *is-basis* (*set-of-vector* X) **using** *exists-basis* **by** *blast*
obtain $Y::'a\hat{}'m\hat{}'m$ **where** Y: *is-basis* (*set-of-vector* Y) **using** *exists-basis* **by** *blast*
obtain $P\ Q$ **where** *B-PAQ*: B=(*matrix-inv* P)$**A**$$*Q$ **and** *inv-P*: *invertible* P **and** *inv-Q*: *invertible* Q
using *equivalent* **unfolding** *equivalent-matrices-def* **by** *auto*
obtain f **where** *f-A*: *matrix'* $X\ Y\ f = A$ **and** *linear-f*: *linear* $((*s))$ $((*s))$ f
using *exists-linear-eq-matrix'*[*OF* $X\ Y$] **by** *auto*
obtain $X'::'a\hat{}'n\hat{}'n$ **where** X': *is-basis* (*set-of-vector* X') **and** Q:*matrix-change-of-basis* $X'\ X = Q$
using *invertible-matrix-is-change-of-basis*[*OF inv-Q* X] **by** *fast*
obtain $Y'::'a\hat{}'m\hat{}'m$ **where** Y': *is-basis* (*set-of-vector* Y') **and** P: *matrix-change-of-basis* $Y'\ Y = P$
using *invertible-matrix-is-change-of-basis*[*OF inv-P* Y] **by** *fast*
have *matrix-inv-P*: *matrix-change-of-basis* $Y\ Y' =$ *matrix-inv* P
using *matrix-inv-matrix-change-of-basis*[*OF* $Y'\ Y$] P **by** *simp*
have *matrix'* $X'\ Y'\ f =$ *matrix-change-of-basis* $Y\ Y'$ $**$ *matrix'* $X\ Y\ f$ $**$ *matrix-change-of-basis* $X'\ X$
using *matrix'-matrix-change-of-basis*[*OF* $X\ X'\ Y\ Y'$ *linear-f*] **.**
also have ... = (*matrix-inv* P) $**$ A $**$ Q **unfolding** *matrix-inv-P f-A* Q **..**
also have ... = B **using** *B-PAQ* **..**
finally show *?thesis* **using** *f-A* $X\ X'\ Y\ Y'$ *linear-f* **by** *fast*
qed

lemma *exist-matrix'-implies-equivalent*:
assumes A: *matrix'* $X\ Y\ f = A$
and B: *matrix'* $X'\ Y'\ f = B$
and X: *is-basis* (*set-of-vector* X)
and Y: *is-basis* (*set-of-vector* Y)
and X': *is-basis* (*set-of-vector* X')
and Y': *is-basis* (*set-of-vector* Y')
and *linear-f*: *linear* $((*s))$ $((*s))$ f
shows *equivalent-matrices* A B
proof (*unfold equivalent-matrices-def*, *rule exI*[*of* - *matrix-change-of-basis* $Y'\ Y$], *rule exI*[*of* - *matrix-change-of-basis* $X'\ X$], *auto*)

have *inv*: *matrix-change-of-basis Y Y' = matrix-inv* (*matrix-change-of-basis Y' Y*) **using** *matrix-inv-matrix-change-of-basis*[*OF Y' Y*] **.**
show *invertible* (*matrix-change-of-basis Y' Y*) **using** *invertible-matrix-change-of-basis*[*OF Y' Y*] **.**
show *invertible* (*matrix-change-of-basis X' X*) **using** *invertible-matrix-change-of-basis*[*OF X' X*] **.**
have *B = matrix' X' Y' f* **using** *B* **..**
also have *... = matrix-change-of-basis Y Y' ** matrix' X Y f ** matrix-change-of-basis X' X* **using** *matrix'-matrix-change-of-basis*[*OF X X' Y Y' linear-f*] **.**
finally show *B = matrix-inv* (*matrix-change-of-basis Y' Y*) *** *A ** matrix-change-of-basis X' X* **unfolding** *inv* **unfolding** *A* **.**
qed

This is the proof of the theorem 2.18 in the book "Advanced Linear Algebra" by Steven Roman.

corollary *equivalent-iff-exist-matrix'*:
shows *equivalent-matrices A B* $\longleftrightarrow$ ($\exists$ *X Y X' Y' f*::'*a*::{*field*}^'*n*$\Rightarrow$'*a*^'*m*.
linear ((∗*s*)) ((∗*s*)) *f* $\wedge$ *matrix' X Y f = A* $\wedge$ *matrix' X' Y' f = B*
$\wedge$ *is-basis* (*set-of-vector X*) $\wedge$ *is-basis* (*set-of-vector Y*)
$\wedge$ *is-basis* (*set-of-vector X'*) $\wedge$ *is-basis* (*set-of-vector Y'*))
by (*rule*, *auto simp add*: *exist-matrix'-implies-equivalent equivalent-implies-exist-matrix'*)

7.7 Similar matrices

definition *similar-matrices* :: '*a*::{*semiring-1*}^'*n*^'*n* $\Rightarrow$ '*a*::{*semiring-1*}^'*n*^'*n* $\Rightarrow$ *bool*
where *similar-matrices A B* = ($\exists$ *P*. *invertible P* $\wedge$ *B*=(*matrix-inv P*)**A**P)

lemma *similar-implies-exist-matrix'*:
fixes *A B*::'*a*::{*field*}^'*n*^'*n*
assumes *similar*: *similar-matrices A B*
shows $\exists$ *X Y f*. *linear* ((∗*s*)) ((∗*s*)) *f* $\wedge$ *matrix' X X f = A* $\wedge$ *matrix' Y Y f = B*
$\wedge$ *is-basis* (*set-of-vector X*) $\wedge$ *is-basis* (*set-of-vector Y*)
proof −
obtain *P* **where** *inv-P*: *invertible P* **and** *B-PAP*: *B*=(*matrix-inv P*)**A**P **using** *similar* **unfolding** *similar-matrices-def* **by** *blast*
obtain *X*::'*a*^'*n*^'*n* **where** *X*: *is-basis* (*set-of-vector X*) **using** *exists-basis* **by** *blast*
obtain *f* **where** *linear-f*: *linear* ((∗*s*)) ((∗*s*)) *f* **and** *A*: *matrix' X X f = A* **using** *exists-linear-eq-matrix'*[*OF X X*] **by** *blast*
obtain *Y*::'*a*^'*n*^'*n* **where** *Y*: *is-basis* (*set-of-vector Y*) **and** *P*: *P = matrix-change-of-basis Y X*
using *invertible-matrix-is-change-of-basis*[*OF inv-P X*] **by** *fast*
have *P'*: *matrix-inv P = matrix-change-of-basis X Y* **by** (*metis* (*lifting*) *P X Y matrix-inv-matrix-change-of-basis*)
have *B* = (*matrix-inv P*)**A**P **using** *B-PAP* **.**
also have *... = matrix-change-of-basis X Y ** matrix' X X f ** P* **unfolding** *P' A* **..**

also have ... = *matrix-change-of-basis X Y ** matrix' X X f ** matrix-change-of-basis Y X* **unfolding** *P* **..**
also have ... = *matrix' Y Y f* **using** *matrix'-matrix-change-of-basis*[*OF X Y X Y linear-f*] **by** *simp*
finally show *?thesis* **using** *X Y A linear-f* **by** *fast*
qed

lemma *exist-matrix'-implies-similar*:
fixes *A B*::'*a*::{*field*}^'*n*^'*n*
assumes *linear-f*: *linear* ((∗*s*)) ((∗*s*)) *f* **and** *A*: *matrix' X X f* = *A* **and** *B*: *matrix' Y Y f* = *B*
and *X*: *is-basis* (*set-of-vector X*) **and** *Y*: *is-basis* (*set-of-vector Y*)
shows *similar-matrices A B*
proof (*unfold similar-matrices-def*, *rule exI*[*of - matrix-change-of-basis Y X*], *rule conjI*)
have *B*=*matrix' Y Y f* **using** *B* **..**
also have ... = *matrix-change-of-basis X Y ** matrix' X X f ** matrix-change-of-basis Y X* **using** *matrix'-matrix-change-of-basis*[*OF X Y X Y linear-f*] **by** *simp*
also have ... = *matrix-inv* (*matrix-change-of-basis Y X*) ** *A* ** *matrix-change-of-basis Y X* **unfolding** *A matrix-inv-matrix-change-of-basis*[*OF Y X*] **..**
finally show *B* = *matrix-inv* (*matrix-change-of-basis Y X*) ** *A* ** *matrix-change-of-basis Y X* **.**
show *invertible* (*matrix-change-of-basis Y X*) **using** *invertible-matrix-change-of-basis*[*OF Y X*] **.**
qed

This is the proof of the theorem 2.19 in the book "Advanced Linear Algebra" by Steven Roman.

corollary *similar-iff-exist-matrix'*:
fixes *A B*::'*a*::{*field*}^'*n*^'*n*
shows *similar-matrices A B* ⟷ (∃ *X Y f*. *linear* ((∗*s*)) ((∗*s*)) *f* ∧ *matrix' X X f* = *A*
∧ *matrix' Y Y f* = *B* ∧ *is-basis* (*set-of-vector X*) ∧ *is-basis* (*set-of-vector Y*))
by (*rule*, *auto simp add*: *exist-matrix'-implies-similar similar-implies-exist-matrix'*)

end

8 Obtaining explicitly the invertible matrix which transforms a matrix to its reduced row echelon form

theory *Gauss-Jordan-PA*
imports
Gauss-Jordan
Rank-Nullity-Theorem.Miscellaneous
Linear-Maps
begin

8.1 Definitions

The following algorithm is similar to *Gauss-Jordan*, but in this case we will also return the P matrix which makes *Gauss-Jordan A = P ** A*. If A is invertible, this matrix P will be the inverse of it.

definition *Gauss-Jordan-in-ij-PA* :: (('a::{*semiring-1*, *inverse*, *one*, *uminus*}^'rows::{*finite*, *ord*}^'rows::{*finite*, *ord*}) × ('a^'cols^'rows::{*finite*, *ord*})) => 'rows=>'cols
=>(('a^'rows::{*finite*, *ord*}^'rows::{*finite*, *ord*}) × ('a^'cols^'rows::{*finite*, *ord*}))
where *Gauss-Jordan-in-ij-PA A' i j* = (*let P=fst A'*; *A=snd A'*;
n = (*LEAST n. A $ n $ j* ≠ *0* ∧ *i* ≤ *n*);
interchange-A = (*interchange-rows A i n*);
interchange-P = (*interchange-rows P i n*);
P' = *mult-row interchange-P i* (*1*/*interchange-Aij*)
in
(*vec-lambda*(% *s. if s=i then P' $ s else* (*row-add P' s i* (−(*interchange-Asj*))) $ *s*), *Gauss-Jordan-in-ij A i j*))

definition *Gauss-Jordan-column-k-PA*
where *Gauss-Jordan-column-k-PA A' k* =
(*let P* = *fst A'*;
i = *fst* (*snd A'*);
A = *snd* (*snd A'*);
from-nat-i=from-nat i;
from-nat-k=from-nat k
in
if (∀ *m*≥*from-nat-i. A $ m $ from-nat-k* = *0*) ∨ *i* = *nrows A then* (*P*, *i*, *A*)
else (*let Gauss* = *Gauss-Jordan-in-ij-PA* (*P*,*A*) (*from-nat-i*) (*from-nat-k*)
in (*fst Gauss*, *i* + *1*, *snd Gauss*)))

definition *Gauss-Jordan-upt-k-PA A k* = (*let foldl=*(*foldl Gauss-Jordan-column-k-PA* (*mat 1*,*0*, *A*) [*0*..<*Suc k*]) *in* (*fst foldl*, *snd* (*snd foldl*)))
definition *Gauss-Jordan-PA A* = *Gauss-Jordan-upt-k-PA A* (*ncols A* − *1*)

8.2 Proofs

8.2.1 Properties about *Gauss-Jordan-in-ij-PA*

The following lemmas are very important in order to improve the efficience of the code

We define the following function to obtain an efficient code for *Gauss-Jordan-in-ij-PA A i j*.

definition *Gauss-Jordan-wrapper i j A B* = *vec-lambda*(%*s. if s=i then A $ s else* (*row-add A s i* (−(*Bsj*))) $ *s*)

lemma *Gauss-Jordan-wrapper-code*[*code abstract*]:
vec-nth (*Gauss-Jordan-wrapper i j A B*) = (%*s. if s=i then A $ s else* (*row-add A s i* (−(*Bsj*))) $ *s*)

unfolding *Gauss-Jordan-wrapper-def* **by** *force*

lemma *Gauss-Jordan-in-ij-PA-def'*[*code*]:
Gauss-Jordan-in-ij-PA A' i j = (*let P*=*fst A'*; *A*=*snd A'*;
n = (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*);
interchange-A = (*interchange-rows A i n*);
A' = *mult-row interchange-A i* (*1*/*interchange-A*$*i*$*j*);
interchange-P = (*interchange-rows P i n*);
P' = *mult-row interchange-P i* (*1*/*interchange-A*$*i*$*j*)
in
(*Gauss-Jordan-wrapper i j P' interchange-A*,
Gauss-Jordan-wrapper i j A' interchange-A))
unfolding *Gauss-Jordan-in-ij-PA-def Gauss-Jordan-in-ij-def Let-def Gauss-Jordan-wrapper-def*
by *auto*

The second component is equal to *Gauss-Jordan-in-ij*

lemma *snd-Gauss-Jordan-in-ij-PA-eq*[*code-unfold*]: *snd* (*Gauss-Jordan-in-ij-PA* (*P*,*A*) *i j*) = *Gauss-Jordan-in-ij A i j*
unfolding *Gauss-Jordan-in-ij-PA-def Let-def snd-conv* **..**

lemma *fst-Gauss-Jordan-in-ij-PA*:
fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *PB-A*: *P* ** *B* = *A*
shows *fst* (*Gauss-Jordan-in-ij-PA* (*P*,*A*) *i j*) ** *B* = *snd* (*Gauss-Jordan-in-ij-PA* (*P*,*A*) *i j*)
proof (*unfold Gauss-Jordan-in-ij-PA-def' Gauss-Jordan-wrapper-def Let-def fst-conv snd-conv*, *subst* (*1 2 3 4 5 6 7 8 9 10*) *interchange-rows-mat-1*[*symmetric*], *subst vec-eq-iff*, *auto*)
show ((χ *s*. *if s* = *i then mult-row* (*interchange-rows* (*mat 1*) *i* (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) ** *P*) *i* (*1* / (*interchange-rows* (*mat 1*) *i* (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) ** *A*) $ *i* $ *j*) $ *s*
else row-add (*mult-row* (*interchange-rows* (*mat 1*) *i* (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) ** *P*) *i* (*1* / (*interchange-rows* (*mat 1*) *i* (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) ** *A*) $ *i* $ *j*)) *s i*
(− (*interchange-rows* (*mat 1*) *i* (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) ** *A*) $ *s* $ *j*) $ *s*) ** *B*) $ *i* =
mult-row (*interchange-rows* (*mat 1*) *i* (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) ** *A*) *i* (*1* / (*interchange-rows* (*mat 1*) *i* (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) ** *A*) $ *i* $ *j*) $ *i*
proof (*unfold matrix-matrix-mult-def*, *vector*, *auto*)
fix *ia*
have *mult-row* (*interchange-rows* (*mat 1*) *i* (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) ** *P*) *i* (*1* / (*interchange-rows* (*mat 1*) *i* (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) ** *A*) $ *i* $ *j*)
** *B* = *mult-row* (*interchange-rows* (*mat 1*) *i* (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) ** *A*) *i* (*1* / (*interchange-rows* (*mat 1*) *i* (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) ** *A*) $ *i* $ *j*)
by(*subst* (*5*) *PB-A*[*symmetric*], *subst* (*1 2*) *mult-row-mat-1*[*symmetric*], *unfold matrix-mul-assoc*, *rule refl*)

thus ($\sum k{\in}UNIV.$ *mult-row* (χ *ia ja.* $\sum k{\in}UNIV.$ *interchange-rows* (*mat 1*) *i* (*LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n*) $ *ia* $ *k* ∗ *P* $ *k* $ *ja*) *i*
(*1 / (*$\sum k{\in}UNIV.$ *mat 1* $ (*LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n*) $ *k* ∗ *A* $ *k* $ *j*)) $ *i* $ *k* ∗ *B* $ *k* $ *ia*) =
mult-row (χ *ia ja.* $\sum k{\in}UNIV.$ *interchange-rows* (*mat 1*) *i* (*LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n*) $ *ia* $ *k* ∗ *A* $ *k* $ *ja*) *i*
(*1 / (*$\sum k{\in}UNIV.$ *mat 1* $ (*LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n*) $ *k* ∗ *A* $ *k* $ *j*)) $ *i* $ *ia*
unfolding *matrix-matrix-mult-def*
unfolding *vec-lambda-beta* **unfolding** *interchange-rows-i* **using** *sum.cong*
by (*metis* (*lifting*, *no-types*) *vec-lambda-beta*)
qed
next
fix *ia* **assume** *ia-not-i*: *ia ≠ i*
have ((χ *s. if s = i then mult-row* (*interchange-rows* (*mat 1*) *i* (*LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n*) ∗∗ *P*) *i* (*1 / (interchange-rows* (*mat 1*) *i* (*LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n*) ∗∗ *A*) $ *i* $ *j*) $
s else row-add (*mult-row* (*interchange-rows* (*mat 1*) *i* (*LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n*) ∗∗ *P*) *i* (*1 / (interchange-rows* (*mat 1*) *i* (*LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n*) ∗∗ *A*) $ *i* $ *j*)) *s*
i (− (*interchange-rows* (*mat 1*) *i* (*LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n*) ∗∗ *A*) $ *s* $ *j*) $ *s*) ∗∗ *B*) $ *ia* =
((χ *s. row-add* (*mult-row* (*interchange-rows* (*mat 1*) *i* (*LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n*) ∗∗ *P*) *i* (*1 / (interchange-rows* (*mat 1*) *i* (*LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n*) ∗∗ *A*) $ *i* $ *j*)) *s*
i (− (*interchange-rows* (*mat 1*) *i* (*LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n*) ∗∗ *A*) $ *s* $ *j*) $ *s*) ∗∗ *B*) $ *ia*
unfolding *row-matrix-matrix-mult*[*symmetric*]
using *ia-not-i* **by** *auto*
also have *... = row-add* (*mult-row* (*interchange-rows* (*mat 1*) *i* (*LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n*) ∗∗ *P*) *i* (*1 / (interchange-rows* (*mat 1*) *i* (*LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n*) ∗∗ *A*) $ *i* $ *j*)) *ia i*
(− (*interchange-rows* (*mat 1*) *i* (*LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n*) ∗∗ *A*) $ *ia* $ *j*) $ *ia v*∗ *B*
by (*subst* (*3*) *row-matrix-matrix-mult*[*symmetric*], *simp*)
also have *... = row-add* (*mult-row* (*interchange-rows* (*mat 1*) *i* (*LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n*) ∗∗ *A*) *i* (*1 / (interchange-rows* (*mat 1*) *i* (*LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n*) ∗∗ *A*) $ *i* $ *j*)) *ia i*
(− (*interchange-rows* (*mat 1*) *i* (*LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n*) ∗∗ *A*) $ *ia* $ *j*) $ *ia*
apply (*subst* (*7*) *PB-A*[*symmetric*])
apply (*subst* (*1 2*) *mult-row-mat-1*[*symmetric*])
apply (*subst* (*1 2*) *row-add-mat-1*[*symmetric*])
unfolding *matrix-mul-assoc*
unfolding *row-matrix-matrix-mult* **..**
finally show ((χ *s. if s = i then mult-row* (*interchange-rows* (*mat 1*) *i* (*LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n*) ∗∗ *P*) *i* (*1 / (interchange-rows* (*mat 1*) *i* (*LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n*) ∗∗ *A*) $ *i* $ *j*) $ *s*
else row-add (*mult-row* (*interchange-rows* (*mat 1*) *i* (*LEAST n. A $ n $ j*

$\neq 0 \wedge i \leq n) ** P)\ i\ (1\ /\ (interchange\text{-}rows\ (mat\ 1)\ i\ (LEAST\ n.\ A\ \$\ n\ \$\ j \neq 0 \wedge i \leq n) ** A)\ \$\ i\ \$\ j))\ s\ i$
$(-\ (interchange\text{-}rows\ (mat\ 1)\ i\ (LEAST\ n.\ A\ \$\ n\ \$\ j \neq 0 \wedge i \leq n) ** A)\ \$\ s\ \$\ j)\ \$\ s) ** B)\ \$\ ia =$
$row\text{-}add\ (mult\text{-}row\ (interchange\text{-}rows\ (mat\ 1)\ i\ (LEAST\ n.\ A\ \$\ n\ \$\ j \neq 0 \wedge i \leq n) ** A)\ i\ (1\ /\ (interchange\text{-}rows\ (mat\ 1)\ i\ (LEAST\ n.\ A\ \$\ n\ \$\ j \neq 0 \wedge i \leq n) ** A)\ \$\ i\ \$\ j))\ ia\ i$
$(-\ (interchange\text{-}rows\ (mat\ 1)\ i\ (LEAST\ n.\ A\ \$\ n\ \$\ j \neq 0 \wedge i \leq n) ** A)\ \$\ ia\ \$\ j)\ \$\ ia$ **.**
qed

8.2.2 Properties about *Gauss-Jordan-column-k-PA*

lemma *fst-Gauss-Jordan-column-k*:
assumes $i \leq nrows\ A$
shows $fst\ (Gauss\text{-}Jordan\text{-}column\text{-}k\ (i,\ A)\ k) \leq nrows\ A$
using *assms* **unfolding** *Gauss-Jordan-column-k-def Let-def* **by** *auto*

lemma *fst-Gauss-Jordan-column-k-PA*:
fixes $A::'a::\{field\}\hat{}'cols::\{mod\text{-}type\}\hat{}'rows::\{mod\text{-}type\}$
assumes *PB-A*: $P ** B = A$
shows $fst\ (Gauss\text{-}Jordan\text{-}column\text{-}k\text{-}PA\ (P,i,A)\ k) ** B = snd\ (snd\ (Gauss\text{-}Jordan\text{-}column\text{-}k\text{-}PA\ (P,i,A)\ k))$
unfolding *Gauss-Jordan-column-k-PA-def* **unfolding** *Let-def*
unfolding *fst-conv snd-conv* **by** (*auto intro*: *assms fst-Gauss-Jordan-in-ij-PA*)

lemma *snd-snd-Gauss-Jordan-column-k-PA-eq*:
shows $snd\ (snd\ (Gauss\text{-}Jordan\text{-}column\text{-}k\text{-}PA\ (P,i,A)\ k)) = snd\ (Gauss\text{-}Jordan\text{-}column\text{-}k\ (i,A)\ k)$
unfolding *Gauss-Jordan-column-k-PA-def Gauss-Jordan-column-k-def* **unfolding** *Let-def snd-conv fst-conv* **unfolding** *snd-Gauss-Jordan-in-ij-PA-eq* **by** *auto*

lemma *fst-snd-Gauss-Jordan-column-k-PA-eq*:
shows $fst\ (snd\ (Gauss\text{-}Jordan\text{-}column\text{-}k\text{-}PA\ (P,i,A)\ k)) = fst\ (Gauss\text{-}Jordan\text{-}column\text{-}k\ (i,A)\ k)$
unfolding *Gauss-Jordan-column-k-PA-def Gauss-Jordan-column-k-def* **unfolding** *Let-def snd-conv fst-conv* **by** *auto*

8.2.3 Properties about *Gauss-Jordan-upt-k-PA*

lemma *fst-Gauss-Jordan-upt-k-PA*:
fixes $A::'a::\{field\}\hat{}'cols::\{mod\text{-}type\}\hat{}'rows::\{mod\text{-}type\}$
shows $fst\ (Gauss\text{-}Jordan\text{-}upt\text{-}k\text{-}PA\ A\ k) ** A = snd\ (Gauss\text{-}Jordan\text{-}upt\text{-}k\text{-}PA\ A\ k)$
proof (*induct k*)
show $fst\ (Gauss\text{-}Jordan\text{-}upt\text{-}k\text{-}PA\ A\ 0) ** A = snd\ (Gauss\text{-}Jordan\text{-}upt\text{-}k\text{-}PA\ A\ 0)$
unfolding *Gauss-Jordan-upt-k-PA-def Let-def fst-conv snd-conv*
apply *auto* **unfolding** *snd-snd-Gauss-Jordan-column-k-PA-eq* **by** (*metis fst-Gauss-Jordan-column-k-PA matrix-mul-lid snd-snd-Gauss-Jordan-column-k-PA-eq*)
next

case (*Suc k*)
have *suc-rw*: [*0*..<*Suc* (*Suc k*)] = [*0*..<*Suc k*] @ [*Suc k*] **by** *simp*
show *?case*
unfolding *Gauss-Jordan-upt-k-PA-def Let-def fst-conv snd-conv*
unfolding *suc-rw* **unfolding** *foldl-append* **unfolding** *List.foldl.simps* **using** *Suc.hyps*[*unfolded Gauss-Jordan-upt-k-PA-def Let-def fst-conv snd-conv*]
by (*metis fst-Gauss-Jordan-column-k-PA prod.collapse*)
qed

lemma *snd-foldl-Gauss-Jordan-column-k-eq*:
snd (*foldl Gauss-Jordan-column-k-PA* (*mat 1*, *0*, *A*) [*0*..<*k*]) = *foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*k*]
proof (*induct k*)
case *0*
show *?case* **by** *simp*
case (*Suc k*)
have *suc-rw*: [*0*..<*Suc k*] = [*0*..<*k*] @ [*k*] **by** *simp*
show *?case*
unfolding *suc-rw foldl-append* **unfolding** *List.foldl.simps* **by** (*metis Suc.hyps fst-snd-Gauss-Jordan-column-k-snd-snd-Gauss-Jordan-column-k-PA-eq surjective-pairing*)
qed

lemma *snd-Gauss-Jordan-upt-k-PA*:
shows *snd* (*Gauss-Jordan-upt-k-PA A k*) = (*Gauss-Jordan-upt-k A k*)
unfolding *Gauss-Jordan-upt-k-PA-def Gauss-Jordan-upt-k-def Let-def*
using *snd-foldl-Gauss-Jordan-column-k-eq*[*of A Suc k*] **by** *simp*

8.2.4 Properties about *Gauss-Jordan-PA*

lemma *fst-Gauss-Jordan-PA*:
fixes *A*::*'a*::{*field*}^*cols*::{*mod-type*}^*rows*::{*mod-type*}
shows *fst* (*Gauss-Jordan-PA A*) ** *A* = *snd* (*Gauss-Jordan-PA A*)
unfolding *Gauss-Jordan-PA-def* **using** *fst-Gauss-Jordan-upt-k-PA* **by** *simp*

lemma *Gauss-Jordan-PA-eq*:
shows *snd* (*Gauss-Jordan-PA A*)= (*Gauss-Jordan A*)
by (*metis Gauss-Jordan-PA-def Gauss-Jordan-def snd-Gauss-Jordan-upt-k-PA*)

8.2.5 Proving that the transformation has been carried out by means of elementary operations

This function is very similar to *row-add-iterate* one. It allows us to prove that *fst* (*Gauss-Jordan-PA A*) is an invertible matrix. Concretly, it has been defined to demonstrate that *fst* (*Gauss-Jordan-PA A*) has been obtained by means of elementary operations applied to the identity matrix

fun *row-add-iterate-PA* :: (('*a*::{*semiring-1*, *uminus*}^*m*::{*mod-type*} ^*m*::{*mod-type*}) × ('*a*^*n*^*m*::{*mod-type*}))=> *nat* => '*m* => '*n* =>
(('*a*^*m*::{*mod-type*} ^*m*::{*mod-type*}) × ('*a*^*n*^*m*::{*mod-type*}))

where *row-add-iterate-PA* (*P*,*A*) *0 i j* = (*if i=0 then* (*P*,*A*) *else* (*row-add P 0 i* (−*A* $ *0* $ *j*), *row-add A 0 i* (−*A* $ *0* $ *j*)))
| *row-add-iterate-PA* (*P*,*A*) (*Suc n*) *i j* = (*if* (*Suc n* = *to-nat i*) *then row-add-iterate-PA* (*P*,*A*) *n i j*
else row-add-iterate-PA ((*row-add P* (*from-nat* (*Suc n*)) *i* (− *A* $ (*from-nat* (*Suc n*)) $ *j*)), (*row-add A* (*from-nat* (*Suc n*)) *i* (− *A* $ (*from-nat* (*Suc n*)) $ *j*))) *n i j*)

lemma *fst-row-add-iterate-PA-preserves-greater-than-n*:
assumes *n*: *n*<*nrows A*
and *a*: *to-nat a* > *n*
shows *fst* (*row-add-iterate-PA* (*P*,*A*) *n i j*) $ *a* $ *b* = *P* $ *a* $ *b*
using *assms*
proof (*induct n arbitrary*: *A P*)
case *0*
show *?case* **unfolding** *row-add-iterate.simps*
proof (*auto*)
assume *i* ≠ *0*
hence *a* ≠ *0* **by** (*metis 0.prems*(*2*) *less-numeral-extra*(*3*) *to-nat-0*)
thus *row-add P 0 i* (− *A* $ *0* $ *j*) $ *a* $ *b* = *P* $ *a* $ *b* **unfolding** *row-add-def* **by** *auto*
qed
next
case (*Suc n*)
have *row-add-iterate-A*: *fst* (*row-add-iterate-PA* (*P*,*A*) *n i j*) $ *a* $ *b* = *P* $ *a* $ *b* **using** *Suc.hyps Suc.prems* **by** *auto*
show *?case*
proof (*cases Suc n* = *to-nat i*)
case *True*
show *fst* (*row-add-iterate-PA* (*P*, *A*) (*Suc n*) *i j*) $ *a* $ *b* = *P* $ *a* $ *b* **unfolding** *row-add-iterate-PA.simps if-P*[*OF True*] **using** *row-add-iterate-A* **.**
next
case *False*
define *A′* **where** *A′* = *row-add A* (*from-nat* (*Suc n*)) *i* (− *A* $ *from-nat* (*Suc n*) $ *j*)
define *P′* **where** *P′* = *row-add P* (*from-nat* (*Suc n*)) *i* (− *A* $ *from-nat* (*Suc n*) $ *j*)
have *row-add-iterate-A′*: *fst* (*row-add-iterate-PA* (*P′*,*A′*) *n i j*) $ *a* $ *b* = *P′* $ *a* $ *b* **using** *Suc.hyps Suc.prems* **unfolding** *nrows-def* **by** *auto*
have *from-nat-not-a*: *from-nat* (*Suc n*) ≠ *a* **by** (*metis less-not-refl Suc.prems to-nat-from-nat-id nrows-def*)
show *fst* (*row-add-iterate-PA* (*P*, *A*) (*Suc n*) *i j*) $ *a* $ *b* = *P* $ *a* $ *b* **unfolding** *row-add-iterate-PA.simps if-not-P*[*OF False*] *row-add-iterate-A′*[*unfolded A′-def P′-def*]
unfolding *row-add-def* **using** *from-nat-not-a* **by** *simp*
qed
qed

lemma *snd-row-add-iterate-PA-eq-row-add-iterate*:
shows *snd* (*row-add-iterate-PA* (*P*,*A*) *n i j*) = *row-add-iterate A n i j*
proof (*induct n arbitrary*: *P A*)
case *0*
show *?case* **unfolding** *row-add-iterate-PA.simps row-add-iterate.simps* **by** *simp*
next
case (*Suc n*)
show *?case* **unfolding** *row-add-iterate-PA.simps row-add-iterate.simps* **by** (*simp add*: *Suc.hyps*)
qed

lemma *row-add-iterate-PA-preserves-pivot-row*:
assumes *n*: *n*<*nrows A*
and *a*: *to-nat i* ≤ *n*
shows *fst* (*row-add-iterate-PA* (*P*,*A*) *n i j*) $ *i* $ *b* = *P* $ *i* $ *b*
using *assms*
proof (*induct n arbitrary*: *P A*)
case *0*
show *?case* **by** (*metis 0.prems*(*2*) *fst-conv le-0-eq row-add-iterate-PA.simps*(*1*) *to-nat-eq-0*)
next
case (*Suc n*)
show *?case*
proof (*cases Suc n* = *to-nat i*)
case *True* **show** *?thesis* **unfolding** *row-add-iterate-PA.simps if-P*[*OF True*]
proof (*rule fst-row-add-iterate-PA-preserves-greater-than-n*)
show *n* < *nrows A* **by** (*metis Suc.prems*(*1*) *Suc-lessD*)
show *n* < *to-nat i* **by** (*metis True lessI*)
qed
next
case *False*
define *P*′ **where** *P*′ = *row-add P* (*from-nat* (*Suc n*)) *i* (− *A* $ *from-nat* (*Suc n*) $ *j*)
define *A*′ **where** *A*′ = *row-add A* (*from-nat* (*Suc n*)) *i* (− *A* $ *from-nat* (*Suc n*) $ *j*)
have *from-nat-noteq-i*: *from-nat* (*Suc n*) ≠ *i* **using** *False Suc.prems*(*1*) *from-nat-not-eq* **unfolding** *nrows-def* **by** *blast*
have *hyp*: *fst* (*row-add-iterate-PA* (*P*′, *A*′) *n i j*) $ *i* $ *b* = *P*′ $ *i* $ *b*
proof (*rule Suc.hyps*)
show *n* < *nrows A*′ **using** *Suc.prems*(*1*) **unfolding** *nrows-def* **by** *simp*
show *to-nat i* ≤ *n* **using** *Suc.prems*(*2*) *False* **by** *simp*
qed
show *?thesis* **unfolding** *row-add-iterate-PA.simps* **unfolding** *if-not-P*[*OF False*] **unfolding** *hyp*[*unfolded A*′*-def P*′*-def*] **unfolding** *row-add-def* **using** *from-nat-noteq-i* **by** *auto*
qed
qed

lemma *fst-row-add-iterate-PA-eq-row-add*:
fixes *A*::*'a*::{*ring-1*}^*'n*^*'m*::{*mod-type*}
assumes *a-not-i*: $a \neq i$
and *n*: $n<nrows\ A$
and *to-nat* $a \leq n$
shows *fst* (*row-add-iterate-PA* (*P*,*A*) *n i j*) $ *a* $ *b* = (*row-add P a i* (− *A* $ *a* $ *j*)) $ *a* $ *b*
using *assms*
proof (*induct n arbitrary*: *A P*)
case *0* **show** *?case* **by** (*metis 0.prems*(*3*) *a-not-i fst-conv le-0-eq row-add-iterate-PA.simps*(*1*) *to-nat-eq-0*)
next
case (*Suc n*)
show *?case*
proof (*cases* *Suc n* = *to-nat i*)
case *True*
show *?thesis*
unfolding *row-add-iterate-PA.simps if-P*[*OF True*]
proof (*rule Suc.hyps*[*OF a-not-i*])
show $n < nrows\ A$ **by** (*metis Suc.prems*(*2*) *Suc-lessD*)
show *to-nat* $a \leq n$ **by** (*metis Suc.prems*(*3*) *True a-not-i le-SucE to-nat-eq*)
qed
next
case *False* **note** *Suc-n-not-i=False*
show *?thesis*
proof (*cases to-nat a* = *Suc n*)
case *True*
show *fst* (*row-add-iterate-PA* (*P*, *A*) (*Suc n*) *i j*) $ *a* $ *b* = *row-add P a i* (− *A* $ *a* $ *j*) $ *a* $ *b*
unfolding *row-add-iterate-PA.simps if-not-P*[*OF False*]
by (*metis Suc-le-lessD True order-refl less-imp-le fst-row-add-iterate-PA-preserves-greater-than-n Suc.prems*(*2*) *to-nat-from-nat nrows-def*)
next
case *False*
define *A'* **where** *A'* = *row-add A* (*from-nat* (*Suc n*)) *i* (− *A* $ *from-nat* (*Suc n*) $ *j*)
define *P'* **where** *P'* = *row-add P* (*from-nat* (*Suc n*)) *i* (− *A* $ *from-nat* (*Suc n*) $ *j*)
have *rw*: *fst* (*row-add-iterate-PA* (*P'*,*A'*) *n i j*) $ *a* $ *b* = *row-add P' a i* (− *A'* $ *a* $ *j*) $ *a* $ *b*
proof (*rule Suc.hyps*)
show $a \neq i$ **using** *Suc.prems*(*1*) **by** *simp*
show $n < nrows\ A'$ **using** *Suc.prems*(*2*) **unfolding** *nrows-def* **by** *auto*
show *to-nat* $a \leq n$ **using** *False Suc.prems*(*3*) **by** *simp*
qed

have *rw1*: *P'* $ *a* $ *b* = *P* $ *a* $ *b*
unfolding *P'-def row-add-def* **using** *False Suc.prems* **unfolding** *nrows-def*

by (*auto simp add*: *to-nat-from-nat-id*)
have *rw2*: $A'\ \$\ a\ \$\ j = A\ \$\ a\ \$\ j$
unfolding *A'-def row-add-def* **using** *False Suc.prems* **unfolding** *nrows-def*
by (*auto simp add*: *to-nat-from-nat-id*)
have *rw3*: $P'\ \$\ i\ \$\ b = P\ \$\ i\ \$\ b$
unfolding *P'-def row-add-def* **using** *False Suc.prems Suc-n-not-i* **unfolding** *nrows-def* **by** (*auto simp add*: *to-nat-from-nat-id*)
show *fst* (*row-add-iterate-PA* (*P*, *A*) (*Suc n*) *i j*) $\ \$\ a\ \$\ b = row\text{-}add\ P\ a\ i\ (-\ A\ \$\ a\ \$\ j)\ \$\ a\ \$\ b$
unfolding *row-add-iterate-PA.simps if-not-P*[*OF Suc-n-not-i*] **unfolding** *rw*[*unfolded P'-def A'-def*]
unfolding *A'-def*[*symmetric*] *P'-def*[*symmetric*] **unfolding** *row-add-def* **apply** *auto*
unfolding *rw1 rw2 rw3* **..**
qed
qed
qed

lemma *fst-row-add-iterate-PA-eq-fst-Gauss-Jordan-in-ij-PA*:
fixes *A*::'*a*::{*field*}^'*cols*::{*mod-type*}^'*rows*::{*mod-type*}
and *i*::'*rows* **and** *j*::'*cols*
and *P*::'*a*::{*field*}^'*rows*::{*mod-type*}^'*rows*::{*mod-type*}
defines *A'*: $A' == mult\text{-}row\ (interchange\text{-}rows\ A\ i\ (LEAST\ n.\ A\ \$\ n\ \$\ j \neq 0 \wedge i \leq n))\ i\ (1\ /\ (interchange\text{-}rows\ A\ i\ (LEAST\ n.\ A\ \$\ n\ \$\ j \neq 0 \wedge i \leq n))\ \$\ i\ \$\ j)$
defines *P'*: $P' == mult\text{-}row\ (interchange\text{-}rows\ P\ i\ (LEAST\ n.\ A\ \$\ n\ \$\ j \neq 0 \wedge i \leq n))\ i\ (1\ /\ (interchange\text{-}rows\ A\ i\ (LEAST\ n.\ A\ \$\ n\ \$\ j \neq 0 \wedge i \leq n))\ \$\ i\ \$\ j)$
shows *fst* (*row-add-iterate-PA* (*P'*,*A'*) (*nrows A* − *1*) *i j*) = *fst* (*Gauss-Jordan-in-ij-PA* (*P*,*A*) *i j*)
proof (*unfold Gauss-Jordan-in-ij-PA-def Let-def*, *vector*, *auto*)
fix *ia*
have *interchange-rw*: $interchange\text{-}rows\ A\ i\ (LEAST\ n.\ A\ \$\ n\ \$\ j \neq 0 \wedge i \leq n)\ \$\ i\ \$\ j = A\ \$\ (LEAST\ n.\ A\ \$\ n\ \$\ j \neq 0 \wedge i \leq n)\ \$\ j$
using *interchange-rows-j*[*symmetric*, *of A* (*LEAST n. A* $\$\ n\ \$\ j \neq 0 \wedge i \leq n$)] **by** *auto*
show *fst* (*row-add-iterate-PA* (*P'*, *A'*) (*nrows A* − *Suc 0*) *i j*) $\ \$\ i\ \$\ ia =$
$mult\text{-}row\ (interchange\text{-}rows\ P\ i\ (LEAST\ n.\ A\ \$\ n\ \$\ j \neq 0 \wedge i \leq n))\ i\ (1\ /\ A\ \$\ (LEAST\ n.\ A\ \$\ n\ \$\ j \neq 0 \wedge i \leq n)\ \$\ j)\ \$\ i\ \$\ ia$
unfolding *A' P' interchange-rw*
proof (*rule row-add-iterate-PA-preserves-pivot-row*, *unfold nrows-def*)
show *CARD*('*rows*) − *Suc 0* < *CARD*('*rows*) **by** *auto*
show *to-nat i* ≤ *CARD*('*rows*) − *Suc 0* **by** (*metis Suc-pred leD not-less-eq-eq to-nat-less-card zero-less-card-finite*)
qed
next
fix *ia iaa*
have *interchange-rw*: $A\ \$\ (LEAST\ n.\ A\ \$\ n\ \$\ j \neq 0 \wedge i \leq n)\ \$\ j = inter\text{-}$

change-rows A i (*LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n*) $ *i* $ *j*
using *interchange-rows-j*[*symmetric*, *of A* (*LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n*)]
by *auto*
assume *ia-not-i*: *ia ≠ i*
have *rw*: (− *interchange-rows A i* (*LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n*) $ *ia* $ *j*)
= − *mult-row* (*interchange-rows A i* (*LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n*)) *i* (*1 / interchange-rows A i* (*LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n*) $ *i* $ *j*) $ *ia* $ *j*
unfolding *interchange-rows-def mult-row-def* **using** *ia-not-i* **by** *auto*
show *fst* (*row-add-iterate-PA* (*P′*, *A′*) (*nrows A − Suc 0*) *i j*) $ *ia* $ *iaa*
= *row-add* (*mult-row* (*interchange-rows P i* (*LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n*)) *i* (*1 / A $* (*LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n*) $ *j*)) *ia i*
(− *interchange-rows A i* (*LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n*) $ *ia* $ *j*) $ *ia* $ *iaa* **unfolding** *interchange-rw* **unfolding** *A′ P′* **unfolding** *rw*
proof (*rule fst-row-add-iterate-PA-eq-row-add*, *unfold nrows-def*)
show *ia ≠ i* **using** *ia-not-i* **.**
show *CARD*(*′rows*) − *Suc 0 < CARD*(*′rows*) **using** *zero-less-card-finite* **by** *auto*
show *to-nat ia ≤ CARD*(*′rows*) − *Suc 0* **by** (*metis Suc-pred leD not-less-eq-eq to-nat-less-card zero-less-card-finite*)
qed
qed

lemma *invertible-fst-row-add-iterate-PA*:
fixes *A*::*′a*::{*ring-1*}^*′n*^*′m*::{*mod-type*}
assumes *n*: *n*<*nrows A*
and *inv-P*: *invertible P*
shows *invertible* (*fst* (*row-add-iterate-PA* (*P*,*A*) *n i j*))
using *n inv-P*
proof (*induct n arbitrary*: *A P*)
case *0*
show *?case*
proof (*unfold row-add-iterate-PA.simps*, *auto simp add*: *0.prems*)
assume *i-not-0*: *i ≠ 0*
have *row-add P 0 i* (− *A $ 0 $ j*) = *row-add* (*mat 1*) *0 i* (− *A $ 0 $ j*) ** *P* **unfolding** *row-add-mat-1* **..**
show *invertible* (*row-add P 0 i* (− *A $ 0 $ j*))
by (*subst row-add-mat-1*[*symmetric*], *rule invertible-mult*, *auto simp add*: *invertible-row-add*[*of 0 i* (− *A $ 0 $ j*)] *i-not-0 0.prems*)
qed
next
case (*Suc n*)
show *?case*
proof (*cases Suc n = to-nat i*)
case *True*
show *?thesis* **unfolding** *row-add-iterate-PA.simps if-P*[*OF True*] **using** *Suc.hyps Suc.prems* **by** *simp*
next
case *False*
show *?thesis*

proof (*unfold row-add-iterate-PA.simps if-not-P*[*OF False*], *rule Suc.hyps*, *unfold nrows-def*)
show $n < CARD('m)$ **using** *Suc.prems*(*1*) **unfolding** *nrows-def* **by** *simp*
show *invertible* (*row-add P* (*from-nat* (*Suc n*)) *i* (− *A* $ *from-nat* (*Suc n*) $ *j*))
proof (*subst row-add-mat-1*[*symmetric*], *rule invertible-mult*, *rule invertible-row-add*)
show *from-nat* (*Suc n*) ≠ *i* **using** *False Suc.prems*(*1*) *from-nat-not-eq* **unfolding** *nrows-def* **by** *blast*
show *invertible P* **using** *Suc.prems*(*2*) **.**
qed
qed
qed
qed

lemma *invertible-fst-Gauss-Jordan-in-ij-PA*:
fixes *A*::'*a*::{*field*}^'*n*::{*mod-type*}^'*m*::{*mod-type*}
assumes *inv-P*: *invertible P*
and *not-all-zero*: ¬ (∀ *m*≥*i*. *A* $ *m* $ *j* = *0*)
shows *invertible* (*fst* (*Gauss-Jordan-in-ij-PA* (*P*,*A*) *i j*))
proof (*unfold fst-row-add-iterate-PA-eq-fst-Gauss-Jordan-in-ij-PA*[*symmetric*], *rule invertible-fst-row-add-iterate-PA*, *simp add*: *nrows-def*,
subst interchange-rows-mat-1[*symmetric*], *subst mult-row-mat-1*[*symmetric*], *rule invertible-mult*)
show *invertible* (*mult-row* (*mat 1*) *i* (*1* / *interchange-rows A i* (*LEAST n*. *A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) $ *i* $ *j*))
proof (*rule invertible-mult-row'*)
have *interchange-rows A i* (*LEAST n*. *A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) $ *i* $ *j* = *A* $ (*LEAST n*. *A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) $ *j* **by** *simp*
also have ... ≠ *0* **by** (*metis* (*lifting*, *mono-tags*) *LeastI-ex not-all-zero*)
finally show *1* / *interchange-rows A i* (*LEAST n*. *A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) $ *i* $ *j* ≠ *0*
unfolding *inverse-eq-divide*[*symmetric*] **using** *nonzero-imp-inverse-nonzero* **by** *blast*
qed
show *invertible* (*interchange-rows* (*mat 1*) *i* (*LEAST n*. *A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) ** *P*)
by (*rule invertible-mult*, *rule invertible-interchange-rows*, *rule inv-P*)
qed

lemma *invertible-fst-Gauss-Jordan-column-k-PA*:
fixes *A*::'*a*::{*field*}^'*n*::{*mod-type*}^'*m*::{*mod-type*}
assumes *inv-P*: *invertible P*
shows *invertible* (*fst* (*Gauss-Jordan-column-k-PA* (*P*,*i*,*A*) *k*))
proof (*unfold Gauss-Jordan-column-k-PA-def Let-def snd-conv fst-conv*, *auto simp add*: *inv-P*)

fix *m*
assume *i-less-m*: *from-nat i* ≤ *m* **and** *Amk-not-0*: *A* $ *m* $ *from-nat k* ≠ *0*
show *invertible* (*fst* (*Gauss-Jordan-in-ij-PA* (*P*, *A*) (*from-nat i*) (*from-nat k*)))
by (*rule invertible-fst-Gauss-Jordan-in-ij-PA*[*OF inv-P*], *auto intro*!: *i-less-m Amk-not-0*)
qed

lemma *invertible-fst-Gauss-Jordan-upt-k-PA*:
fixes *A*::*'a*::{*field*}^*n*::{*mod-type*}^*m*::{*mod-type*}
shows *invertible* (*fst* (*Gauss-Jordan-upt-k-PA A k*))
proof (*induct k*)
case *0*
show *?case* **unfolding** *Gauss-Jordan-upt-k-PA-def Let-def fst-conv* **by** (*simp add*: *invertible-fst-Gauss-Jordan-column-k-PA invertible-mat-1*)
next
case (*Suc k*)
have *list-rw*: [*0*..<*Suc* (*Suc k*)] = [*0*..<*Suc k*] @ [*Suc k*] **by** *simp*
define *f* **where** *f* = *foldl Gauss-Jordan-column-k-PA* (*mat 1*, *0*, *A*) [*0*..<*Suc k*]
have *f-rw*: *f* = (*fst f*, *fst* (*snd f*), *snd* (*snd f*)) **by** *simp*
show *?case* **unfolding** *Gauss-Jordan-upt-k-PA-def Let-def fst-conv*
unfolding *list-rw* **unfolding** *foldl-append* **unfolding** *List.foldl.simps* **using** *invertible-fst-Gauss-Jordan-column-k-PA*
by (*metis* (*mono-tags*) *Gauss-Jordan-upt-k-PA-def Suc.hyps fst-conv prod.collapse*)
qed

lemma *invertible-fst-Gauss-Jordan-PA*:
fixes *A*::*'a*::{*field*}^*n*::{*mod-type*}^*m*::{*mod-type*}
shows *invertible* (*fst* (*Gauss-Jordan-PA A*))
by (*unfold Gauss-Jordan-PA-def*, *rule invertible-fst-Gauss-Jordan-upt-k-PA*)

definition *P-Gauss-Jordan A* = *fst* (*Gauss-Jordan-PA A*)

end

9 Computing determinants of matrices using the Gauss Jordan algorithm

theory *Determinants2*
imports
Gauss-Jordan-PA
begin

9.1 Some previous properties

9.1.1 Relationships between determinants and elementary row operations

lemma *det-interchange-rows*:
shows *det* (*interchange-rows A i j*) = *of-int* (*if i* = *j then 1 else* −*1*) ∗ *det A*
proof −

have (*interchange-rows A i j*) = (χ *a*. *A* $ (*Transposition.transpose i j*) *a*)
unfolding *interchange-rows-def Transposition.transpose-def* **by** *vector*
hence *det*(*interchange-rows A i j*) = *det*(χ *a*. *A*$(*Transposition.transpose i j*) *a*)
by *simp*
also have ... = *of-int* (*sign* (*Transposition.transpose i j*)) ∗ *det A* **by** (*rule det-permute-rows*[*of Transposition.transpose i j A*], *simp add*: *permutes-swap-id*)
finally show *?thesis* **unfolding** *sign-swap-id* **.**
qed

corollary *det-interchange-different-rows*:
assumes *i-not-j*: *i* ≠ *j*
shows *det* (*interchange-rows A i j*) = − *det A* **unfolding** *det-interchange-rows*
using *i-not-j* **by** *simp*

corollary *det-interchange-same-rows*:
assumes *i-eq-j*: *i* = *j*
shows *det* (*interchange-rows A i j*) = *det A* **unfolding** *det-interchange-rows* **using** *i-eq-j* **by** *simp*

lemma *det-mult-row*:
shows *det* (*mult-row A a k*) = *k* ∗ *det A*
proof −
have *A-rw*: (χ *i*. *if i* = *a then A*$*a else A*$*i*) = *A* **by** *vector*
have (*mult-row A a k*) = (χ *i*. *if i* = *a then k* ∗*s A* $ *a else A* $ *i*) **unfolding** *mult-row-def* **by** *vector*
hence *det*(*mult-row A a k*) = *det*(χ *i*. *if i* = *a then k* ∗*s A* $ *a else A* $ *i*) **by** *simp*
also have ... = *k* ∗ *det*(χ *i*. *if i* = *a then A*$*a else A*$*i*) **unfolding** *det-row-mul*
..
also have ... = *k* ∗ *det A* **unfolding** *A-rw* **..**
finally show *?thesis* **.**
qed

lemma *det-row-add′*:
assumes *i-not-j*: *i* ≠ *j*
shows *det* (*row-add A i j q*) = *det A*
proof −
have (*row-add A i j q*) = (χ *k*. *if k* = *i then row i A* + *q* ∗*s row j A else row k A*)
unfolding *row-add-def row-def* **by** *vector*
hence *det*(*row-add A i j q*) = *det*(χ *k*. *if k* = *i then row i A* + *q* ∗*s row j A else row k A*) **by** *simp*
also have ... = *det A* **unfolding** *det-row-operation*[*OF i-not-j*] **..**
finally show *?thesis* **.**
qed

9.1.2 Relationships between determinants and elementary column operations

lemma *det-interchange-columns*:

shows *det* (*interchange-columns A i j*) = *of-int* (*if i* = *j then 1 else* −*1*) ∗ *det A*
proof −
have (*interchange-columns A i j*) = (χ *a b*. *A* $ *a* $ (*Transposition.transpose i j*) *b*)
unfolding *interchange-columns-def Transposition.transpose-def* **by** *vector*
hence *det*(*interchange-columns A i j*) = *det*(χ *a b*. *A* $ *a* $ (*Transposition.transpose i j*) *b*) **by** *simp*
also have ... = *of-int* (*sign* (*Transposition.transpose i j*)) ∗ *det A* **by** (*rule det-permute-columns*[*of Transposition.transpose i j A*], *simp add*: *permutes-swap-id*)
finally show *?thesis* **unfolding** *sign-swap-id* **.**
qed

corollary *det-interchange-different-columns*:
assumes *i-not-j*: *i* ≠ *j*
shows *det* (*interchange-columns A i j*) = − *det A* **unfolding** *det-interchange-columns*
using *i-not-j* **by** *simp*

corollary *det-interchange-same-columns*:
assumes *i-eq-j*: *i* = *j*
shows *det* (*interchange-columns A i j*) = *det A* **unfolding** *det-interchange-columns*
using *i-eq-j* **by** *simp*

lemma *det-mult-columns*:
shows *det* (*mult-column A a k*) = *k* ∗ *det A*
proof −
have *mult-column A a k* = *transpose* (*mult-row* (*transpose A*) *a k*) **unfolding** *transpose-def mult-row-def mult-column-def* **by** *vector*
hence *det* (*mult-column A a k*) = *det* (*transpose* (*mult-row* (*transpose A*) *a k*)) **by** *simp*
also have ... = *det* (*mult-row* (*transpose A*) *a k*) **unfolding** *det-transpose* **..**
also have ... = *k* ∗ *det* (*transpose A*) **unfolding** *det-mult-row* **..**
also have ... = *k* ∗ *det A* **unfolding** *det-transpose* **..**
finally show *?thesis* **.**
qed

lemma *det-column-add*:
assumes *i-not-j*: *i* ≠ *j*
shows *det* (*column-add A i j q*) = *det A*
proof −
have (*column-add A i j q*) = (*transpose* (*row-add* (*transpose A*) *i j q*)) **unfolding** *transpose-def column-add-def row-add-def* **by** *vector*
hence *det* (*column-add A i j q*) = *det* (*transpose* (*row-add* (*transpose A*) *i j q*)) **by** *simp*
also have ... = *det* (*row-add* (*transpose A*) *i j q*) **unfolding** *det-transpose* **..**
also have ... = *det A* **unfolding** *det-row-add'*[*OF i-not-j*] *det-transpose* **..**
finally show *?thesis* **.**
qed

9.2 Proving that the determinant can be computed by means of the Gauss Jordan algorithm

9.2.1 Previous properties

lemma *det-row-add-iterate-upt-n*:
fixes *A*::*'a*::{*comm-ring-1*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
assumes *n*: *n*<*nrows A*
shows *det* (*row-add-iterate A n i j*) = *det A*
using *n*
proof (*induct n arbitrary*: *A*)
case *0*
show *?case* **unfolding** *row-add-iterate.simps* **using** *det-row-add'*[*of 0 i A*] **by** *auto*
next
case (*Suc n*)
show *?case* **unfolding** *row-add-iterate.simps*
proof (*auto*)
show *det* (*row-add-iterate A n i j*) = *det A* **using** *Suc.hyps Suc.prems* **by** *simp*
assume *Suc-n-not-i*: *Suc n* ≠ *to-nat i*
have *det* (*row-add-iterate* (*row-add A* (*from-nat* (*Suc n*)) *i* (− *A* $ *from-nat* (*Suc n*) $ *j*)) *n i j*)
= *det* (*row-add A* (*from-nat* (*Suc n*)) *i* (− *A* $ *from-nat* (*Suc n*) $ *j*))
proof (*rule Suc.hyps*, *unfold nrows-def*)
show *n* < *CARD*(*'n*) **using** *Suc.prems* **unfolding** *nrows-def* **by** *auto*
qed
also have ... = *det A*
proof (*rule det-row-add'*,*rule ccontr*, *simp*)
assume *from-nat* (*Suc n*) = *i*
hence *to-nat* (*from-nat* (*Suc n*)::*'n*) = *to-nat i* **by** *simp*
hence (*Suc n*) = *to-nat i* **unfolding** *to-nat-from-nat-id*[*OF Suc.prems*[*unfolded nrows-def*]] **.**
thus *False* **using** *Suc-n-not-i* **by** *contradiction*
qed
finally show *det* (*row-add-iterate* (*row-add A* (*from-nat* (*Suc n*)) *i* (− *A* $ *from-nat* (*Suc n*) $ *j*)) *n i j*) = *det A* **.**
qed
qed

corollary *det-row-add-iterate*:
fixes *A*::*'a*::{*comm-ring-1*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
shows *det* (*row-add-iterate A* (*nrows A* − *1*) *i j*) = *det A*
by (*metis det-row-add-iterate-upt-n diff-less neq0-conv nrows-not-0 zero-less-one*)

lemma *det-Gauss-Jordan-in-ij*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*} **and** *i j*::*'n*
defines *A'*: *A'*== *mult-row* (*interchange-rows A i* (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*)) *i* (*1* / (*interchange-rows A i* (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*)) $ *i* $ *j*)

shows *det (Gauss-Jordan-in-ij A i j) = det A'*
proof −
have *nrows-eq*: *nrows A' = nrows A* **unfolding** *nrows-def* **by** *simp*
have *row-add-iterate A' (nrows A − 1) i j = Gauss-Jordan-in-ij A i j* **using** *row-add-iterate-eq-Gauss-Jordan-in-ij* **unfolding** *A'* **.**
hence *det (Gauss-Jordan-in-ij A i j) = det (row-add-iterate A' (nrows A − 1) i j)* **by** *simp*
also have *... = det A'* **by** (*rule det-row-add-iterate*[*of A', unfolded nrows-eq*])
finally show *?thesis* **.**
qed

lemma *det-Gauss-Jordan-in-ij-1*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*} **and** *i j*::*'n*
defines *A'*: *A'*== *mult-row (interchange-rows A i (LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n)) i (1 / (interchange-rows A i (LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n)) $ i $ j)*
assumes *i*: *(LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n) = i*
shows *det (Gauss-Jordan-in-ij A i j) = 1/(Aij) ∗ det A*
proof −
have *det (Gauss-Jordan-in-ij A i j) = det A'* **using** *det-Gauss-Jordan-in-ij* **unfolding** *A'* **by** *auto*
also have *... = 1/(Aij) ∗ det A* **unfolding** *A' det-mult-row* **unfolding** *i det-interchange-rows* **by** *auto*
finally show *?thesis* **.**
qed

lemma *det-Gauss-Jordan-in-ij-2*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*} **and** *i j*::*'n*
defines *A'*: *A'*== *mult-row (interchange-rows A i (LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n)) i (1 / (interchange-rows A i (LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n)) $ i $ j)*
assumes *i*: *(LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n) ≠ i*
shows *det (Gauss-Jordan-in-ij A i j) = − 1/(A $ (LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n) $ j) ∗ det A*
proof −
have *det (Gauss-Jordan-in-ij A i j) = det A'* **using** *det-Gauss-Jordan-in-ij* **unfolding** *A'* **by** *auto*
also have *... = − 1/(A$ (LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n) $j) ∗ det A* **unfolding** *A' det-mult-row* **unfolding** *det-interchange-rows* **using** *i* **by** *auto*
finally show *?thesis* **.**
qed

9.2.2 Definitions

The following definitions allow the computation of the determinant of a matrix using the Gauss-Jordan algorithm. In the first component the determinant of each transformation is accumulated and the second component contains the matrix transformed into a reduced row echelon form matrix

definition *Gauss-Jordan-in-ij-det-P* :: *'a*::{*semiring-1*, *inverse*, *one*, *uminus*}^*'m*^*'n*::{*finite*, *ord*}=> *'n*=>*'m*=>(*'a* × (*'a*^*'m*^*'n*::{*finite*, *ord*}))
where *Gauss-Jordan-in-ij-det-P A i j* = (*let n* = (*LEAST n*. *A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) *in* (*if i* = *n then 1*/(*A* $ *i* $ *j*) *else* − *1*/(*A* $ *n* $ *j*), *Gauss-Jordan-in-ij A i j*))

definition *Gauss-Jordan-column-k-det-P* **where** *Gauss-Jordan-column-k-det-P A' k* =
(*let det-P*= *fst A'*; *i* = *fst* (*snd A'*); *A* = *snd* (*snd A'*); *from-nat-i* = *from-nat i*; *from-nat-k* = *from-nat k*
in if (∀ *m*≥*from-nat-i*. *A* $ *m* $ *from-nat-k* = *0*) ∨ *i* = *nrows A then* (*det-P*, *i*, *A*)
else let gauss = *Gauss-Jordan-in-ij-det-P A* (*from-nat-i*) (*from-nat-k*) *in* (*fst gauss* ∗ *det-P*, *i* + *1*, *snd gauss*))

definition *Gauss-Jordan-upt-k-det-P*
where *Gauss-Jordan-upt-k-det-P A k* = (*let foldl* = *foldl Gauss-Jordan-column-k-det-P* (*1*, *0*, *A*) [*0*..<*Suc k*] *in* (*fst foldl*, *snd* (*snd foldl*)))
definition *Gauss-Jordan-det-P*
where *Gauss-Jordan-det-P A* = *Gauss-Jordan-upt-k-det-P A* (*ncols A* − *1*)

9.2.3 Proofs

This is an equivalent definition created to achieve a more efficient computation.

lemma *Gauss-Jordan-in-ij-det-P-code*[*code*]:
shows *Gauss-Jordan-in-ij-det-P A i j* =
(*let n* = (*LEAST n*. *A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*);
interchange-A = *interchange-rows A i n*;
A' = *mult-row interchange-A i* (*1* / *interchange-A* $ *i* $ *j*) *in* (*if i* = *n then 1*/(*A* $ *i* $ *j*) *else* − *1*/(*A* $ *n* $ *j*), *Gauss-Jordan-wrapper i j A' interchange-A*))
unfolding *Gauss-Jordan-in-ij-det-P-def Gauss-Jordan-in-ij-def Gauss-Jordan-wrapper-def Let-def* **by** *auto*

lemma *det-Gauss-Jordan-in-ij-det-P*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*} **and** *i j*::*'n*
shows (*fst* (*Gauss-Jordan-in-ij-det-P A i j*)) ∗ *det A* = *det* (*snd* (*Gauss-Jordan-in-ij-det-P A i j*))
unfolding *Gauss-Jordan-in-ij-det-P-def Let-def fst-conv snd-conv*
using *det-Gauss-Jordan-in-ij-1*[*of A j i*]
using *det-Gauss-Jordan-in-ij-2*[*of A j i*] **by** *auto*

lemma *det-Gauss-Jordan-column-k-det-P*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
assumes *det*: *det-P* ∗ *det B* = *det A*
shows (*fst* (*Gauss-Jordan-column-k-det-P* (*det-P*,*i*,*A*) *k*)) ∗ *det B* = *det* (*snd* (*snd* (*Gauss-Jordan-column-k-det-P* (*det-P*,*i*,*A*) *k*)))

proof (*unfold Gauss-Jordan-column-k-det-P-def Let-def*, *auto simp add*: *assms*)
fix *m*
assume *i-not-nrows*: $i \neq nrows\ A$
and *i-less-m*: $from\text{-}nat\ i \leq m$
and *Amk-not-0*: $A\ \$\ m\ \$\ from\text{-}nat\ k \neq 0$
show $fst\ (Gauss\text{-}Jordan\text{-}in\text{-}ij\text{-}det\text{-}P\ A\ (from\text{-}nat\ i)\ (from\text{-}nat\ k)) * det\text{-}P * det\ B$
=
$det\ (snd\ (Gauss\text{-}Jordan\text{-}in\text{-}ij\text{-}det\text{-}P\ A\ (from\text{-}nat\ i)\ (from\text{-}nat\ k)))$ **unfolding** *mult.assoc det*
unfolding *det-Gauss-Jordan-in-ij-det-P* **..**
qed

lemma *det-Gauss-Jordan-upt-k-det-P*:
fixes $A::'a::\{field\}\ ^\wedge 'n::\{mod\text{-}type\}\ ^\wedge 'n::\{mod\text{-}type\}$
shows $(fst\ (Gauss\text{-}Jordan\text{-}upt\text{-}k\text{-}det\text{-}P\ A\ k)) * det\ A = det\ (snd\ (Gauss\text{-}Jordan\text{-}upt\text{-}k\text{-}det\text{-}P\ A\ k))$
proof (*induct k*)
case *0*
show *?case*
unfolding *Gauss-Jordan-upt-k-det-P-def Let-def* **unfolding** *fst-conv snd-conv* **by** (*simp add*:*det-Gauss-Jordan-column-k-det-P*)
next
case (*Suc k*)
have *suc-rw*: $[0..<Suc\ (Suc\ k)] = [0..<Suc\ k]\ @\ [Suc\ k]$ **by** *simp*
have *fold-expand*: $(foldl\ Gauss\text{-}Jordan\text{-}column\text{-}k\text{-}det\text{-}P\ (1,\ 0,\ A)\ [0..<Suc\ k])$
$= (fst\ (foldl\ Gauss\text{-}Jordan\text{-}column\text{-}k\text{-}det\text{-}P\ (1,\ 0,\ A)\ [0..<Suc\ k]),\ fst\ (snd\ (foldl\ Gauss\text{-}Jordan\text{-}column\text{-}k\text{-}det\text{-}P\ (1,\ 0,\ A)\ [0..<Suc\ k])),$
$snd\ (snd\ (foldl\ Gauss\text{-}Jordan\text{-}column\text{-}k\text{-}det\text{-}P\ (1,\ 0,\ A)\ [0..<Suc\ k])))$ **by** *simp*
show *?case* **unfolding** *Gauss-Jordan-upt-k-det-P-def Let-def*
unfolding *suc-rw foldl-append List.foldl.simps fst-conv snd-conv*
by(*subst* (*1 2*) *fold-expand*, *rule det-Gauss-Jordan-column-k-det-P*, *rule Suc.hyps*[*unfolded Gauss-Jordan-upt-k-det-P-def Let-def fst-conv snd-conv*])
qed

lemma *det-Gauss-Jordan-det-P*:
fixes $A::'a::\{field\}\ ^\wedge 'n::\{mod\text{-}type\}\ ^\wedge 'n::\{mod\text{-}type\}$
shows $(fst\ (Gauss\text{-}Jordan\text{-}det\text{-}P\ A)) * det\ A = det\ (snd\ (Gauss\text{-}Jordan\text{-}det\text{-}P\ A))$
using *det-Gauss-Jordan-upt-k-det-P* **unfolding** *Gauss-Jordan-det-P-def* **by** *simp*

definition *upper-triangular-upt-k* **where** $upper\text{-}triangular\text{-}upt\text{-}k\ A\ k = (\forall\ i\ j.\ j<i \wedge to\text{-}nat\ j < k \longrightarrow A\ \$\ i\ \$\ j = 0)$
definition *upper-triangular* **where** $upper\text{-}triangular\ A = (\forall\ i\ j.\ j<i \longrightarrow A\ \$\ i\ \$\ j = 0)$

lemma *upper-triangular-upt-imp-upper-triangular*:
assumes *upper-triangular-upt-k A* (*nrows A*)

shows *upper-triangular A*
using *assms* **unfolding** *upper-triangular-upt-k-def upper-triangular-def nrows-def*
using *to-nat-less-card*[**where** *?'a='b*] **by** *blast*

lemma *rref-imp-upper-triagular-upt*:
fixes *A*::*'a*::{*one*, *zero*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
assumes *reduced-row-echelon-form A*
shows *upper-triangular-upt-k A k*
proof (*induct k*)
case *0*
show *?case* **unfolding** *upper-triangular-upt-k-def* **by** *simp*
next
case (*Suc k*)
show *?case* **unfolding** *upper-triangular-upt-k-def* **proof** (*clarify*)
fix *i j*::*'n*
assume *j-less-i*: *j* < *i* **and** *j-less-suc-k*: *to-nat j* < *Suc k*
show *A* $ *i* $ *j* = *0*
 proof (*cases to-nat j* < *k*)
 case *True*
 thus *?thesis* **using** *Suc.hyps* **unfolding** *upper-triangular-upt-k-def* **using** *j-less-i True* **by** *auto*
 next
 case *False*
 hence *j-eq-k*: *to-nat j* = *k* **using** *j-less-suc-k* **by** *simp*
 have *rref-suc*: *reduced-row-echelon-form-upt-k A* (*Suc k*) **by** (*metis assms rref-implies-rref-upt*)

 show *?thesis*
 proof (*cases A* $ *i* $ *from-nat k* = *0*)
 case *True*
 have *from-nat k* = *j* **by** (*metis from-nat-to-nat-id j-eq-k*)
 thus *?thesis* **using** *True* **by** *simp*
 next
 case *False*
 have *zero-i-k*: *is-zero-row-upt-k i k A* **unfolding** *is-zero-row-upt-k-def*
 by (*metis* (*opaque-lifting*, *mono-tags*) *Suc.hyps leD le-less-linear less-imp-le j-eq-k j-less-i le-trans to-nat-mono' upper-triangular-upt-k-def*)
 have *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *A* **unfolding** *is-zero-row-upt-k-def* **using** *False* **by** (*metis j-eq-k lessI to-nat-from-nat*)
 have *Least-eq*: (*LEAST n*. *A* $ *i* $ *n* ≠ *0*) = *from-nat k*
 proof (*rule Least-equality*)
 show *A* $ *i* $ *from-nat k* ≠ *0* **using** *False* **by** *simp*
 show ⋀*y*. *A* $ *i* $ *y* ≠ *0* ⟹ *from-nat k* ≤ *y* **by** (*metis* (*full-types*) *is-zero-row-upt-k-def not-le-imp-less to-nat-le zero-i-k*)
 qed
 have *i-not-k*: *i* ≠ *from-nat k* **by** (*metis less-irrefl from-nat-to-nat-id j-eq-k j-less-i*)
 show *?thesis* **using** *rref-upt-condition4-explicit*[*OF rref-suc not-zero-i-suc-k i-not-k*] **unfolding** *Least-eq*
 using *rref-upt-condition1-explicit*[*OF rref-suc*]

using *Suc.hyps* **unfolding** *upper-triangular-upt-k-def*
by (*metis* (*mono-tags*) *leD not-le-imp-less is-zero-row-upt-k-def is-zero-row-upt-k-suc j-eq-k j-less-i not-zero-i-suc-k to-nat-from-nat to-nat-mono'*)
qed
qed
qed
qed

lemma *rref-imp-upper-triagular*:
assumes *reduced-row-echelon-form A*
shows *upper-triangular A*
by (*metis assms rref-imp-upper-triagular-upt upper-triangular-upt-imp-upper-triangular*)

lemma *det-Gauss-Jordan*[*code-unfold*]:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
shows *det* (*Gauss-Jordan A*) = *prod* (λ*i*. (*Gauss-Jordan A*)$*i*$*i*) (*UNIV*:: *'n set*)
using *det-upperdiagonal rref-imp-upper-triagular*[*OF rref-Gauss-Jordan*[*of A*]] **unfolding** *upper-triangular-def* **by** *blast*

lemma *snd-Gauss-Jordan-in-ij-det-P-is-snd-Gauss-Jordan-in-ij-PA*:
shows *snd* (*Gauss-Jordan-in-ij-det-P A i j*) = *snd* (*Gauss-Jordan-in-ij-PA* (*P*,*A*) *i j*)
unfolding *Gauss-Jordan-in-ij-det-P-def Gauss-Jordan-in-ij-PA-def*
unfolding *Gauss-Jordan-in-ij-def Let-def snd-conv fst-conv* **..**

lemma *snd-Gauss-Jordan-column-k-det-P-is-snd-Gauss-Jordan-column-k-PA*:
shows *snd* (*Gauss-Jordan-column-k-det-P* (*n*,*i*,*A*) *k*) = *snd* (*Gauss-Jordan-column-k-PA* (*P*,*i*,*A*) *k*)
unfolding *Gauss-Jordan-column-k-det-P-def Gauss-Jordan-column-k-PA-def Let-def snd-conv* **unfolding** *fst-conv*
using *snd-Gauss-Jordan-in-ij-det-P-is-snd-Gauss-Jordan-in-ij-PA* **by** *auto*

lemma *det-fst-row-add-iterate-PA*:
fixes *A*::*'a*::{*comm-ring-1*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
assumes *n*: *n*<*nrows A*
shows *det* (*fst* (*row-add-iterate-PA* (*P*,*A*) *n i j*)) = *det P*
using *n*
proof (*induct n arbitrary*: *P A*)
case *0*
show *?case* **unfolding** *row-add-iterate-PA.simps* **using** *det-row-add'*[*of 0 i P*] **by** *simp*
next
case (*Suc n*)
have *n*: *n*<*nrows A* **using** *Suc.prems* **by** *simp*
show *?case*

proof (*cases Suc n = to-nat i*)
case *True* **show** *?thesis* **unfolding** *row-add-iterate-PA.simps if-P*[*OF True*] **using** *Suc.hyps*[*OF n*] **.**
next
case *False*
define P' **where** $P' =$ *row-add P* (*from-nat* (*Suc n*)) *i* (− *A* $ *from-nat* (*Suc n*) $ *j*)
define A' **where** $A' =$ *row-add A* (*from-nat* (*Suc n*)) *i* (− *A* $ *from-nat* (*Suc n*) $ *j*)
have *n2*: *n*< *nrows* A' **using** *n* **unfolding** *nrows-def* **.**
have *det* (*fst* (*row-add-iterate-PA* (*P*, *A*) (*Suc n*) *i j*)) = *det* (*fst* (*row-add-iterate-PA* (P', A') *n i j*)) **unfolding** *row-add-iterate-PA.simps if-not-P*[*OF False*] P'*-def* A'*-def* **..**
also have ... = *det* P' **using** *Suc.hyps*[*OF n2*] **.**
also have ... = *det P* **unfolding** P'*-def*
proof (*rule det-row-add'*, *rule ccontr*, *simp*)
assume *from-nat* (*Suc n*) = *i*
hence *to-nat* (*from-nat* (*Suc n*)::*'n*) = *to-nat i* **by** *simp*
hence (*Suc n*) = *to-nat i* **unfolding** *to-nat-from-nat-id*[*OF Suc.prems*[*unfolded nrows-def*]] **.**
thus *False* **using** *False* **by** *contradiction*
qed
finally show *?thesis* **.**
qed
qed

lemma *det-fst-Gauss-Jordan-in-ij-PA-eq-fst-Gauss-Jordan-in-ij-det-P*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
shows *fst* (*Gauss-Jordan-in-ij-det-P A i j*) ∗ *det P* = *det* (*fst* (*Gauss-Jordan-in-ij-PA* (*P*,*A*) *i j*))
proof −
define P' **where** $P' =$ *mult-row* (*interchange-rows P i* (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*)) *i* (*1 / interchange-rows A i* (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) $ *i* $ *j*)
define A' **where** $A' =$ *mult-row* (*interchange-rows A i* (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*)) *i* (*1 / interchange-rows A i* (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) $ *i* $ *j*)
have *det* (*fst* (*Gauss-Jordan-in-ij-PA* (*P*,*A*) *i j*)) = *det* (*fst* (*row-add-iterate-PA* (P',A') (*nrows A* − *1*) *i j*))
unfolding *fst-row-add-iterate-PA-eq-fst-Gauss-Jordan-in-ij-PA*[*symmetric*] A'*-def* P'*-def* **..**
also have ...= *det* P' **by** (*rule det-fst-row-add-iterate-PA*, *simp add*: *nrows-def*)
also have ... = (*if i* = (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) *then 1 / A* $ *i* $ *j else* − *1 / A* $ (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) $ *j*) ∗ *det P*
proof (*cases i* = (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*))
case *True* **show** *?thesis*
unfolding *if-P*[*OF True*] P'*-def* **unfolding** *True*[*symmetric*] **unfolding** *interchange-same-rows* **unfolding** *det-mult-row* **..**
next

case *False*
show *?thesis* **unfolding** *if-not-P*[*OF False*] *P'-def* **unfolding** *det-mult-row* **unfolding** *det-interchange-different-rows*[*OF False*] **by** *simp*
qed
also have ... = *fst* (*Gauss-Jordan-in-ij-det-P A i j*) ∗ *det P*
unfolding *Gauss-Jordan-in-ij-det-P-def* **by** *simp*
finally show *?thesis* **..**
qed

lemma *det-fst-Gauss-Jordan-column-k-PA-eq-fst-Gauss-Jordan-column-k-det-P*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
shows *fst* (*Gauss-Jordan-column-k-det-P* (*det P*,*i*,*A*) *k*) = *det* (*fst* (*Gauss-Jordan-column-k-PA* (*P*,*i*,*A*) *k*))
unfolding *Gauss-Jordan-column-k-det-P-def Gauss-Jordan-column-k-PA-def Let-def snd-conv fst-conv*
using *det-fst-Gauss-Jordan-in-ij-PA-eq-fst-Gauss-Jordan-in-ij-det-P* **by** *auto*

lemma *fst-snd-Gauss-Jordan-column-k-det-P-eq-fst-snd-Gauss-Jordan-column-k-PA*:
shows *fst* (*snd* (*Gauss-Jordan-column-k-det-P* (*n*,*i*,*A*) *k*)) = *fst* (*snd* (*Gauss-Jordan-column-k-PA* (*P*,*i*,*A*) *k*))
unfolding *Gauss-Jordan-column-k-det-P-def Gauss-Jordan-column-k-PA-def Let-def snd-conv fst-conv*
by *auto*

The way of proving the following lemma is very similar to the demonstration of *?k* < *ncols ?A* ⟹ *reduced-row-echelon-form-upt-k* (*Gauss-Jordan-upt-k ?A ?k*) (*Suc ?k*)

?k < *ncols ?A* ⟹ *foldl Gauss-Jordan-column-k* (*0*, *?A*) [*0*..<*Suc ?k*] = (*if* ∀ *m*. *is-zero-row-upt-k m* (*Suc ?k*) (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *?A*) [*0*..<*Suc ?k*])) *then 0 else mod-type-class.to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n* (*Suc ?k*) (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *?A*) [*0*..<*Suc ?k*]))) + *1*, *snd* (*foldl Gauss-Jordan-column-k* (*0*, *?A*) [*0*..<*Suc ?k*])).

lemma *foldl-Gauss-Jordan-column-k-det-P*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
shows *det-fst-Gauss-Jordan-upt-k-PA-eq-fst-Gauss-Jordan-upt-k-det-P*: *fst* (*Gauss-Jordan-upt-k-det-P A k*) = *det* (*fst* (*Gauss-Jordan-upt-k-PA A k*))
and *snd-Gauss-Jordan-upt-k-det-P-is-snd-Gauss-Jordan-upt-k-PA*: *snd* (*Gauss-Jordan-upt-k-det-P A k*) = *snd* (*Gauss-Jordan-upt-k-PA A k*)
and *fst-snd-foldl-Gauss-det-P-PA*: *fst* (*snd* (*foldl Gauss-Jordan-column-k-det-P* (*1*, *0*, *A*) [*0*..<*Suc k*])) = *fst* (*snd* (*foldl Gauss-Jordan-column-k-PA* (*mat 1*, *0*, *A*) [*0*..<*Suc k*]))
proof (*induct k*)
case *0*
show *fst* (*Gauss-Jordan-upt-k-det-P A 0*) = *det* (*fst* (*Gauss-Jordan-upt-k-PA A 0*))

unfolding *Gauss-Jordan-upt-k-det-P-def Gauss-Jordan-upt-k-PA-def Let-def*
by (*simp*, *metis det-fst-Gauss-Jordan-column-k-PA-eq-fst-Gauss-Jordan-column-k-det-P det-I*)
show *snd* (*Gauss-Jordan-upt-k-det-P A 0*) = *snd* (*Gauss-Jordan-upt-k-PA A 0*)
unfolding *Gauss-Jordan-upt-k-det-P-def Gauss-Jordan-upt-k-PA-def Let-def snd-conv*
apply *auto* **using** *snd-Gauss-Jordan-column-k-det-P-is-snd-Gauss-Jordan-column-k-PA*
by *metis*
show *fst* (*snd* (*foldl Gauss-Jordan-column-k-det-P* (*1*, *0*, *A*) [*0*..<*Suc 0*])) = *fst* (*snd* (*foldl Gauss-Jordan-column-k-PA* (*mat 1*, *0*, *A*) [*0*..<*Suc 0*]))
using [[*unfold-abs-def* = *false*]]
unfolding *Gauss-Jordan-column-k-det-P-def Gauss-Jordan-column-k-PA-def* **apply** *auto*
using *fst-snd-Gauss-Jordan-column-k-det-P-eq-fst-snd-Gauss-Jordan-column-k-PA*
by *metis*
next
fix *k*
assume *hyp1*: *fst* (*Gauss-Jordan-upt-k-det-P A k*) = *det* (*fst* (*Gauss-Jordan-upt-k-PA A k*))
and *hyp2*: *snd* (*Gauss-Jordan-upt-k-det-P A k*) = *snd* (*Gauss-Jordan-upt-k-PA A k*)
and *hyp3*: *fst* (*snd* (*foldl Gauss-Jordan-column-k-det-P* (*1*, *0*, *A*) [*0*..<*Suc k*])) = *fst* (*snd* (*foldl Gauss-Jordan-column-k-PA* (*mat 1*, *0*, *A*) [*0*..<*Suc k*]))
have *list-rw*: [*0*..<*Suc* (*Suc k*)] = [*0*..<*Suc k*] @ [*Suc k*] **by** *simp*
have *det-mat-nn*: *det* (*mat 1*::'*a*^'*n*::{*mod-type*}^'*n*::{*mod-type*}) = *1* **using** *det-I* **by** *simp*
define *f* **where** *f* = *foldl Gauss-Jordan-column-k-det-P* (*1*, *0*, *A*) [*0*..<*Suc k*]
define *g* **where** *g* = *foldl Gauss-Jordan-column-k-PA* (*mat 1*, *0*, *A*) [*0*..<*Suc k*]
have *f-rw*: *f* = (*fst f*, *fst* (*snd f*), *snd*(*snd f*)) **by** *simp*
have *g-rw*: *g* = (*fst g*, *fst* (*snd g*), *snd*(*snd g*)) **by** *simp*
have *fst-snd*: *fst* (*snd f*) = *fst* (*snd g*) **unfolding** *f-def g-def* **using** *hyp3* **unfolding** *Gauss-Jordan-upt-k-det-P-def Gauss-Jordan-upt-k-PA-def Let-def fst-conv snd-conv*
.
have *snd-snd*: *snd* (*snd f*) = *snd* (*snd g*) **unfolding** *f-def g-def* **using** *hyp2* **unfolding** *Gauss-Jordan-upt-k-det-P-def Gauss-Jordan-upt-k-PA-def Let-def fst-conv snd-conv* **.**
have *fst-det*: *fst f* = *det* (*fst g*) **unfolding** *f-def g-def* **using** *hyp1* **unfolding** *Gauss-Jordan-upt-k-det-P-def Gauss-Jordan-upt-k-PA-def Let-def fst-conv* **by** *simp*
show *fst* (*snd* (*foldl Gauss-Jordan-column-k-det-P* (*1*, *0*, *A*) [*0*..<*Suc k*])) = *fst* (*snd* (*foldl Gauss-Jordan-column-k-PA* (*mat 1*, *0*, *A*) [*0*..<*Suc k*])) ⟹
fst (*Gauss-Jordan-upt-k-det-P A* (*Suc k*)) = *det* (*fst* (*Gauss-Jordan-upt-k-PA A* (*Suc k*)))
unfolding *Gauss-Jordan-upt-k-det-P-def*
unfolding *Gauss-Jordan-upt-k-PA-def Let-def fst-conv*
unfolding *list-rw foldl-append* **unfolding** *List.foldl.simps*
unfolding *f-def*[*symmetric*] *g-def*[*symmetric*]
apply (*subst f-rw*)
apply (*subst g-rw*)
unfolding *fst-snd snd-snd fst-det*
by (*rule det-fst-Gauss-Jordan-column-k-PA-eq-fst-Gauss-Jordan-column-k-det-P*)

show *snd* (*Gauss-Jordan-upt-k-det-P A* (*Suc k*)) = *snd* (*Gauss-Jordan-upt-k-PA A* (*Suc k*))
unfolding *Gauss-Jordan-upt-k-det-P-def*
unfolding *Gauss-Jordan-upt-k-PA-def Let-def fst-conv*
unfolding *list-rw foldl-append* **unfolding** *List.foldl.simps*
unfolding *f-def*[*symmetric*] *g-def*[*symmetric*]
apply (*subst f-rw*)
apply (*subst g-rw*)
unfolding *fst-snd snd-snd fst-det*
by (*metis fst-snd prod.collapse snd-Gauss-Jordan-column-k-det-P-is-snd-Gauss-Jordan-column-k-PA snd-eqD snd-snd*)
show *fst* (*snd* (*foldl Gauss-Jordan-column-k-det-P* (*1*, *0*, *A*) [*0*..<*Suc* (*Suc k*)]))
= *fst* (*snd* (*foldl Gauss-Jordan-column-k-PA* (*mat 1*, *0*, *A*) [*0*..<*Suc* (*Suc k*)]))
unfolding *Gauss-Jordan-upt-k-det-P-def*
unfolding *Gauss-Jordan-upt-k-PA-def Let-def fst-conv*
unfolding *list-rw foldl-append* **unfolding** *List.foldl.simps*
unfolding *f-def*[*symmetric*] *g-def*[*symmetric*]
apply (*subst f-rw*)
apply (*subst g-rw*)
unfolding *fst-snd snd-snd fst-det* **by** (*rule fst-snd-Gauss-Jordan-column-k-det-P-eq-fst-snd-Gauss-Jordan-colu*
qed

lemma *snd-Gauss-Jordan-det-P-is-Gauss-Jordan*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
shows *snd* (*Gauss-Jordan-det-P A*) = (*Gauss-Jordan A*)
unfolding *Gauss-Jordan-det-P-def Gauss-Jordan-def* **unfolding** *snd-Gauss-Jordan-upt-k-det-P-is-snd-Gauss-*

snd-Gauss-Jordan-upt-k-PA **..**

lemma *det-snd-Gauss-Jordan-det-P*[*code-unfold*]:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
shows *det* (*snd* (*Gauss-Jordan-det-P A*)) = *prod* (λ*i*. (*snd* (*Gauss-Jordan-det-P A*))$*i*$*i*) (*UNIV*:: *'n set*)
unfolding *snd-Gauss-Jordan-det-P-is-Gauss-Jordan det-Gauss-Jordan* **..**

lemma *det-fst-Gauss-Jordan-PA-eq-fst-Gauss-Jordan-det-P*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
shows *fst* (*Gauss-Jordan-det-P A*) = *det* (*fst* (*Gauss-Jordan-PA A*))
by (*unfold Gauss-Jordan-det-P-def Gauss-Jordan-PA-def*, *rule det-fst-Gauss-Jordan-upt-k-PA-eq-fst-Gauss-Jo*

lemma *fst-Gauss-Jordan-det-P-not-0*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
shows *fst* (*Gauss-Jordan-det-P A*) ≠ *0*
unfolding *det-fst-Gauss-Jordan-PA-eq-fst-Gauss-Jordan-det-P*

by (*metis* (*mono-tags*) *det-I det-mul invertible-fst-Gauss-Jordan-PA matrix-inv-right mult-zero-left zero-neq-one*)

lemma *det-code-equation*[*code-unfold*]:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
shows *det A* = (*let A'* = *Gauss-Jordan-det-P A in prod* (λ*i*. (*snd* (*A'*))$*i*$*i*) (*UNIV*::*'n set*)/(*fst* (*A'*)))
unfolding *Let-def* **using** *det-Gauss-Jordan-det-P*[*of A*]
unfolding *det-snd-Gauss-Jordan-det-P*
by (*simp add*: *fst-Gauss-Jordan-det-P-not-0 nonzero-eq-divide-eq ac-simps*)

end

10 Inverse of a matrix using the Gauss Jordan algorithm

theory *Inverse*
imports
Gauss-Jordan-PA
begin

10.1 Several properties

Properties about Gauss Jordan algorithm, reduced row echelon form, rank, identity matrix and invertibility

lemma *rref-id-implies-invertible*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
assumes *Gauss-mat-1*: *Gauss-Jordan A* = *mat 1*
shows *invertible A*
proof −
obtain *P* **where** *P*: *invertible P* **and** *PA*: *Gauss-Jordan A* = *P* ** *A* **using** *invertible-Gauss-Jordan*[*of A*] **by** *blast*
have *A* = *mat 1* ** *A* **unfolding** *matrix-mul-lid* **..**
also have ... = (*matrix-inv P* ** *P*) ** *A* **using** *P invertible-def matrix-inv-unique* **by** *metis*
also have ... = (*matrix-inv P*) ** (*P* ** *A*) **by** (*metis PA assms calculation matrix-eq matrix-vector-mul-assoc matrix-vector-mul-lid*)
also have ... = (*matrix-inv P*) ** *mat 1* **unfolding** *PA*[*symmetric*] *Gauss-mat-1* **..**
also have ... = (*matrix-inv P*) **unfolding** *matrix-mul-rid* **..**
finally have *A* = (*matrix-inv P*) **.**
thus *?thesis* **using** *P* **unfolding** *invertible-def* **using** *matrix-inv-unique* **by** *blast*
qed

In the following case, nrows is equivalent to ncols due to we are working with a square matrix

lemma *full-rank-implies-invertible*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
assumes *rank-n*: *rank A* = *nrows A*
shows *invertible A*
proof (*unfold invertible-left-inverse*[*of A*] *matrix-left-invertible-ker*, *clarify*)
fix *x*
assume *Ax*: *A* ∗*v x* = *0*
have *rank-eq-card-n*: *rank A* = *CARD*(*'n*) **using** *rank-n* **unfolding** *nrows-def* **.**
have *vec.dim* (*null-space A*)=*0* **unfolding** *dim-null-space* **unfolding** *rank-eq-card-n dimension-vector* **by** *simp*
hence *null-space A* = {*0*} **using** *vec.dim-zero-eq* **using** *Ax null-space-def* **by** *auto*
thus *x* = *0* **unfolding** *null-space-def* **using** *Ax* **by** *blast*
qed

lemma *invertible-implies-full-rank*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
assumes *inv-A*: *invertible A*
shows *rank A* = *nrows A*
proof −
have (∀ *x*. *A* ∗*v x* = *0* ⟶ *x* = *0*) **using** *inv-A* **unfolding** *invertible-left-inverse*[*unfolded matrix-left-invertible-ker*] **.**
hence *null-space-eq-0*: (*null-space A*) = {*0*} **unfolding** *null-space-def* **using** *matrix-vector-mult-0-right* **by** *fast*
have *dim-null-space*: *vec.dim* (*null-space A*) = *0* **unfolding** *vec.dim-def*
by (*metis* (*no-types*) *null-space-eq-0 vec.dim-def vec.dim-zero-eq'*)
show *?thesis* **using** *rank-nullity-theorem-matrices*[*of A*] **unfolding** *dim-null-space rank-eq-dim-col-space nrows-def*
unfolding *col-space-eq* **unfolding** *ncols-def* **by** *simp*
qed

definition *id-upt-k* :: *'a*::{*zero*, *one*}^*'n*::{*mod-type*}^*'n*::{*mod-type*} ⇒ *nat* => *bool*
where *id-upt-k A k* = (∀ *i j*. *to-nat i* < *k* ∧ *to-nat j* < *k* ⟶ ((*i* = *j* ⟶ *A* $ *i* $ *j* = *1*) ∧ (*i* ≠ *j* ⟶ *A* $ *i* $ *j* = *0*)))

lemma *id-upt-nrows-mat-1*:
assumes *id-upt-k A* (*nrows A*)
shows *A* = *mat 1*
unfolding *mat-def* **apply** *vector* **using** *assms* **unfolding** *id-upt-k-def nrows-def*
using *to-nat-less-card*[**where** *?'a*=*'b*]
by *presburger*

10.2 Computing the inverse of a matrix using the Gauss Jordan algorithm

This lemma is essential to demonstrate that the Gauss Jordan form of an invertible matrix is the identity. The proof is made by induction and it is ex-

plained in http://www.unirioja.es/cu/jodivaso/Isabelle/Gauss-Jordan-2013-2-Generalized/Demonstration_invertible.pdf

lemma *id-upt-k-Gauss-Jordan*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
assumes *inv-A*: *invertible A*
shows *id-upt-k* (*Gauss-Jordan A*) *k*
proof (*induct k*)
case *0*
show *?case* **unfolding** *id-upt-k-def* **by** *fast*
next
case (*Suc k*)
note *id-k*=*Suc.hyps*
have *rref-k*: *reduced-row-echelon-form-upt-k* (*Gauss-Jordan A*) *k* **using** *rref-implies-rref-upt*[*OF rref-Gauss-Jordan*] **.**
have *rref-suc-k*: *reduced-row-echelon-form-upt-k* (*Gauss-Jordan A*) (*Suc k*) **using** *rref-implies-rref-upt*[*OF rref-Gauss-Jordan*] **.**
have *inv-gj*: *invertible* (*Gauss-Jordan A*) **by** (*metis inv-A invertible-Gauss-Jordan invertible-mult*)
show *id-upt-k* (*Gauss-Jordan A*) (*Suc k*)
proof (*unfold id-upt-k-def*, *auto*)
fix *j*::*'n*
assume *j-less-suc*: *to-nat j* < *Suc k*
— First of all we prove a property which will be useful later
have *greatest-prop*: *j* ≠ *0* ⟹ *to-nat j* = *k* ⟹ (*GREATEST m*. ¬ *is-zero-row-upt-k m k* (*Gauss-Jordan A*)) = *j* − *1*
proof (*rule Greatest-equality*)
assume *j-not-zero*: *j* ≠ *0* **and** *j-eq-k*: *to-nat j* = *k*
have *j-minus-1*: *to-nat* (*j* − *1*) < *k* **by** (*metis* (*full-types*) *Suc-le'* *diff-add-cancel j-eq-k j-not-zero to-nat-mono*)
show ¬ *is-zero-row-upt-k* (*j* − *1*) *k* (*Gauss-Jordan A*)
unfolding *is-zero-row-upt-k-def*
proof (*auto*, *rule exI*[*of* - *j* − *1*], *rule conjI*)
show *to-nat* (*j* − *1*) < *k* **using** *j-minus-1* **.**
show *Gauss-Jordan A* $ (*j* − *1*) $ (*j* − *1*) ≠ *0* **using** *id-k* **unfolding** *id-upt-k-def* **using** *j-minus-1* **by** *simp*
qed
fix *a*::*'n*
assume *not-zero-a*: ¬ *is-zero-row-upt-k a k* (*Gauss-Jordan A*)
show *a* ≤ *j* − *1*
proof (*rule ccontr*)
assume ¬ *a* ≤ *j* − *1*
hence *a-greater-i-minus-1*: *a* > *j* − *1* **by** *simp*
have *is-zero-row-upt-k a k* (*Gauss-Jordan A*)
unfolding *is-zero-row-upt-k-def*
proof (*clarify*)
fix *b*::*'n* **assume** *a*: *to-nat b* < *k*
have *Least-eq*: (*LEAST n*. *Gauss-Jordan A* $ *b* $ *n* ≠ *0*) = *b*
proof (*rule Least-equality*)
show *Gauss-Jordan A* $ *b* $ *b* ≠ *0* **by** (*metis a id-k id-upt-k-def*

zero-neq-one)
show $\bigwedge y$. *Gauss-Jordan A $ b $ y ≠ 0 ⟹ b ≤ y*
by (*metis* (*opaque-lifting*, *no-types*) *a not-less-iff-gr-or-eq id-k id-upt-k-def less-trans not-less to-nat-mono*)
qed
moreover have ¬ *is-zero-row-upt-k b k* (*Gauss-Jordan A*)
unfolding *is-zero-row-upt-k-def* **apply** *auto* **apply** (*rule exI*[*of - b*]) **using** *a id-k* **unfolding** *id-upt-k-def* **by** *simp*
moreover have *a ≠ b*
proof −
have *b < from-nat k* **by** (*metis a from-nat-to-nat-id j-eq-k not-less-iff-gr-or-eq to-nat-le*)
also have ... = *j* **using** *j-eq-k to-nat-from-nat* **by** *auto*
also have ... ≤ *a* **using** *a-greater-i-minus-1* **by** (*metis diff-add-cancel le-Suc*)
finally show *?thesis* **by** *simp*
qed
ultimately show *Gauss-Jordan A $ a $ b = 0* **using** *rref-upt-condition4*[*OF rref-k*] **by** *auto*
qed
thus *False* **using** *not-zero-a* **by** *contradiction*
qed
qed

show *Gauss-jj-1*: *Gauss-Jordan A $ j $ j = 1*
proof (*cases j=0*)
— In case that j be zero, the result is trivial
case *True* **show** *?thesis*
proof (*unfold True*, *rule rref-first-element*)
show *reduced-row-echelon-form* (*Gauss-Jordan A*) **by** (*rule rref-Gauss-Jordan*)
show *column 0* (*Gauss-Jordan A*) ≠ *0* **by** (*metis det-zero-column inv-gj invertible-det-nz*)
qed
next
case *False* **note** *j-not-zero = False*
show *?thesis*
proof (*cases to-nat j < k*)
case *True* **thus** *?thesis* **using** *id-k* **unfolding** *id-upt-k-def* **by** *presburger* — Easy due to the inductive hypothesis
next
case *False*
hence *j-eq-k*: *to-nat j = k* **using** *j-less-suc* **by** *auto*
have *j-minus-1*: *to-nat* (*j − 1*) *< k* **by** (*metis* (*full-types*) *Suc-le′ diff-add-cancel j-eq-k j-not-zero to-nat-mono*)
have (*GREATEST m*. ¬ *is-zero-row-upt-k m k* (*Gauss-Jordan A*)) = *j − 1* **by** (*rule greatest-prop*[*OF j-not-zero j-eq-k*])
hence *zero-j-k*: *is-zero-row-upt-k j k* (*Gauss-Jordan A*)
by (*metis not-le greatest-ge-nonzero-row j-eq-k j-minus-1 to-nat-mono′*)
show *?thesis*

proof (*rule ccontr*, *cases Gauss-Jordan A $ j $ j = 0*)
case *False*
note *gauss-jj-not-0 = False*
assume *gauss-jj-not-1*: *Gauss-Jordan A $ j $ j ≠ 1*
have (*LEAST n. Gauss-Jordan A $ j $ n ≠ 0*) = *j*
proof (*rule Least-equality*)
show *Gauss-Jordan A $ j $ j ≠ 0* **using** *gauss-jj-not-0* **.**
show ⋀*y*. *Gauss-Jordan A $ j $ y ≠ 0* ⟹ *j* ≤ *y* **by** (*metis le-less-linear is-zero-row-upt-k-def j-eq-k to-nat-mono zero-j-k*)
qed
hence *Gauss-Jordan A $ j $* (*LEAST n. Gauss-Jordan A $ j $ n ≠ 0*) ≠ *1* **using** *gauss-jj-not-1* **by** *auto* — Contradiction with the second condition of rref
thus *False* **by** (*metis gauss-jj-not-0 is-zero-row-upt-k-def j-eq-k lessI rref-suc-k rref-upt-condition2*)
next
case *True*
note *gauss-jj-0 = True*
have *zero-j-suc-k*: *is-zero-row-upt-k j* (*Suc k*) (*Gauss-Jordan A*)
by (*rule is-zero-row-upt-k-suc*[*OF zero-j-k*], *metis gauss-jj-0 j-eq-k to-nat-from-nat*)
have ¬ (∃ *B*. *B* ∗∗ (*Gauss-Jordan A*) = *mat 1*) — This will be a contradiction
proof (*unfold matrix-left-invertible-independent-columns*, *simp*,
rule exI[*of - λi. (if i < j then column j* (*Gauss-Jordan A*) $ *i else if i=j then −1 else 0*)], *rule conjI*)
show (∑ *i*∈*UNIV*. (*if i < j then column j* (*Gauss-Jordan A*) $ *i else if i=j then −1 else 0*) ∗*s column i* (*Gauss-Jordan A*)) = *0*
proof (*unfold vec-eq-iff sum-component*, *auto*)
— We write the column j in a linear combination of the previous ones, which is a contradiction (the matrix wouldn't be invertible)
let *?f=λi. (if i < j then column j* (*Gauss-Jordan A*) $ *i else if i=j then −1 else 0*)
fix *i*
let *?g=*(*λx. ?f x ∗ column x* (*Gauss-Jordan A*) $ *i*)
show *sum ?g UNIV = 0*
proof (*cases i<j*)
case *True* **note** *i-less-j = True*
have *sum-rw*: *sum ?g* (*UNIV* − {*i*}) = *?g j* + *sum ?g* ((*UNIV* − {*i*}) − {*j*})
proof (*rule sum.remove*)
show *finite* (*UNIV* − {*i*}) **using** *finite-code* **by** *simp*
show *j* ∈ *UNIV* − {*i*} **using** *True* **by** *blast*
qed
have *sum-g0*: *sum ?g* (*UNIV* − {*i*} − {*j*}) = *0*
proof (*rule sum.neutral*, *auto*)
fix *a*
assume *a-not-j*: *a* ≠ *j* **and** *a-not-i*: *a* ≠ *i* **and** *a-less-j*: *a* < *j* **and** *column-a-not-zero*: *column a* (*Gauss-Jordan A*) $ *i* ≠ *0*

```
                    have Gauss-Jordan A $ i $ a = 0 using id-k unfolding
id-upt-k-def using a-less-j j-eq-k using i-less-j a-not-i to-nat-mono by blast
                    thus column j (Gauss-Jordan A) $ a = 0 using
column-a-not-zero unfolding column-def by simp — Contradiction
                  qed
                  have sum ?g UNIV = ?g i + sum ?g (UNIV − {i}) by
(rule sum.remove, simp-all)
                   also have ... = ?g i + ?g j + sum ?g (UNIV − {i} −
{j}) unfolding sum-rw by auto
                  also have ... = ?g i + ?g j unfolding sum-g0 by simp
                  also have ... = 0 using True unfolding column-def
                   by (simp, metis id-k id-upt-k-def j-eq-k to-nat-mono)
                  finally show ?thesis .
                  next
                  case False
                 have zero-i-suc-k: is-zero-row-upt-k i (Suc k) (Gauss-Jordan
A)
                    by (metis False zero-j-suc-k linorder-cases rref-suc-k
rref-upt-condition1)
                  show ?thesis
                   proof (rule sum.neutral, auto)
                    show column j (Gauss-Jordan A) $ i = 0
                          using zero-i-suc-k unfolding column-def
is-zero-row-upt-k-def
                     by (metis j-eq-k lessI vec-lambda-beta)
                    next
                    fix a
                      assume a-not-j: a ≠ j and a-less-j: a < j and
column-a-i: column a (Gauss-Jordan A) $ i ≠ 0
                     have Gauss-Jordan A $ i $ a = 0 using zero-i-suc-k
unfolding is-zero-row-upt-k-def
                       by (metis (full-types) a-less-j j-eq-k less-SucI
to-nat-mono)
                     thus column j (Gauss-Jordan A) $ a = 0 using
column-a-i unfolding column-def by simp
                   qed
                 qed
                 qed
                 next
                 show ∃ i. (if i < j then column j (Gauss-Jordan A) $ i else if i =
j then −1 else 0) ≠ 0
               by (metis False j-eq-k neg-equal-0-iff-equal to-nat-mono zero-neq-one)
                 qed
                 thus False using inv-gj unfolding invertible-def by simp
                 qed
                 qed
                qed
                 fix i::'n
                 assume i-less-suc: to-nat i < Suc k and i-not-j: i ≠ j
```

show *Gauss-Jordan A $ i $ j = 0* — This result is proved making use of the 4th condition of rref
proof (*cases to-nat i < k ∧ to-nat j < k*)
case *True* **thus** *?thesis* **using** *id-k i-not-j* **unfolding** *id-upt-k-def* **by** *blast* — Easy due to the inductive hypothesis
next
case *False* **note** *i-or-j-ge-k = False*
show *?thesis*
proof (*cases to-nat i < k*)
case *True*
hence *j-eq-k*: *to-nat j = k* **using** *i-or-j-ge-k j-less-suc* **by** *simp*
have *j-noteq-0*: *j ≠ 0* **by** (*metis True j-eq-k less-nat-zero-code to-nat-0*)
have *j-minus-1*: *to-nat (j − 1) < k* **by** (*metis* (*full-types*) *Suc-le′ diff-add-cancel j-eq-k j-noteq-0 to-nat-mono*)
have (*GREATEST m. ¬ is-zero-row-upt-k m k* (*Gauss-Jordan A*)) *= j − 1* **by** (*rule greatest-prop*[*OF j-noteq-0 j-eq-k*])
hence *zero-j-k*: *is-zero-row-upt-k j k* (*Gauss-Jordan A*)
by (*metis* (*lifting*, *mono-tags*) *less-linear less-asym j-eq-k j-minus-1 not-greater-Greatest to-nat-mono*)
have *Least-eq-j*: (*LEAST n. Gauss-Jordan A $ j $ n ≠ 0*) *= j*
proof (*rule Least-equality*)
show *Gauss-Jordan A $ j $ j ≠ 0* **using** *Gauss-jj-1* **by** *simp*
show ⋀*y. Gauss-Jordan A $ j $ y ≠ 0 ⟹ j ≤ y*
by (*metis True le-cases from-nat-to-nat-id i-or-j-ge-k is-zero-row-upt-k-def j-less-suc less-Suc-eq-le less-le to-nat-le zero-j-k*)
qed
moreover have ¬ *is-zero-row-upt-k j* (*Suc k*) (*Gauss-Jordan A*) **unfolding** *is-zero-row-upt-k-def* **by** (*metis Gauss-jj-1 j-less-suc zero-neq-one*)
ultimately show *?thesis* **using** *rref-upt-condition4*[*OF rref-suc-k*] *i-not-j* **by** *fastforce*
next
case *False*
hence *i-eq-k*: *to-nat i = k* **by** (*metis ‹to-nat i < Suc k›* *less-SucE*)
hence *j-less-k*: *to-nat j < k* **by** (*metis i-not-j j-less-suc less-SucE to-nat-from-nat*)
have (*LEAST n. Gauss-Jordan A $ j $ n ≠ 0*) *= j*
proof (*rule Least-equality*)
show *Gauss-Jordan A $ j $ j ≠ 0* **by** (*metis Gauss-jj-1 zero-neq-one*)
show ⋀*y. Gauss-Jordan A $ j $ y ≠ 0 ⟹ j ≤ y*
by (*metis le-cases id-k id-upt-k-def j-less-k less-trans not-less to-nat-mono*)
qed
moreover have ¬ *is-zero-row-upt-k j k* (*Gauss-Jordan A*) **by** (*metis* (*full-types*) *Gauss-jj-1 is-zero-row-upt-k-def j-less-k zero-neq-one*)

ultimately show *?thesis* **using** *rref-upt-condition4*[*OF rref-k*] *i-not-j* **by** *fastforce*
qed
qed
qed
qed

lemma *invertible-implies-rref-id*:
fixes *A*::*'a*::{*field*}^*n*::{*mod-type*}^*n*::{*mod-type*}
assumes *inv-A*: *invertible A*
shows *Gauss-Jordan A* = *mat 1*
using *id-upt-k-Gauss-Jordan*[*OF inv-A*, *of nrows* (*Gauss-Jordan A*)]
using *id-upt-nrows-mat-1*
by *fast*

lemma *matrix-inv-Gauss*:
fixes *A*::*'a*::{*field*}^*n*::{*mod-type*}^*n*::{*mod-type*}
assumes *inv-A*: *invertible A* **and** *Gauss-eq*: *Gauss-Jordan A* = *P* ** *A*
shows *matrix-inv A* = *P*
proof (*unfold matrix-inv-def*, *rule some1-equality*)
show ∃!*A'*. *A* ** *A'* = *mat 1* ∧ *A'* ** *A* = *mat 1* **by** (*metis inv-A invertible-def matrix-inv-unique matrix-left-right-inverse*)
show *A* ** *P* = *mat 1* ∧ *P* ** *A* = *mat 1* **by** (*metis Gauss-eq inv-A invertible-implies-rref-id matrix-left-right-inverse*)
qed

lemma *matrix-inv-Gauss-Jordan-PA*:
fixes *A*::*'a*::{*field*}^*n*::{*mod-type*}^*n*::{*mod-type*}
assumes *inv-A*: *invertible A*
shows *matrix-inv A* = *fst* (*Gauss-Jordan-PA A*)
by (*metis Gauss-Jordan-PA-eq fst-Gauss-Jordan-PA inv-A matrix-inv-Gauss*)

lemma *invertible-eq-full-rank*[*code-unfold*]:
fixes *A*::*'a*::{*field*}^*n*::{*mod-type*}^*n*::{*mod-type*}
shows *invertible A* = (*rank A* = *nrows A*)
by (*metis full-rank-implies-invertible invertible-implies-full-rank*)

definition *inverse-matrix A* = (*if invertible A then Some* (*matrix-inv A*) *else None*)

lemma *the-inverse-matrix*:
fixes *A*::*'a*::{*field*}^*n*::{*mod-type*}^*n*::{*mod-type*}
assumes *invertible A*
shows *the* (*inverse-matrix A*) = *P-Gauss-Jordan A*
by (*metis P-Gauss-Jordan-def assms inverse-matrix-def matrix-inv-Gauss-Jordan-PA option.sel*)

lemma *inverse-matrix*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
shows *inverse-matrix A* = (*if invertible A then Some* (*P-Gauss-Jordan A*) *else None*)
by (*metis inverse-matrix-def option.sel the-inverse-matrix*)

lemma *inverse-matrix-code*[*code-unfold*]:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
shows *inverse-matrix A* = (*let GJ* = *Gauss-Jordan-PA A*;
rank-A = (*if A* = *0 then 0 else to-nat* (*GREATEST a. row a* (*snd GJ*) ≠ *0*) + *1*) *in*
if nrows A = *rank-A then Some* (*fst*(*GJ*)) *else None*)
unfolding *inverse-matrix*
unfolding *invertible-eq-full-rank*
unfolding *rank-Gauss-Jordan-code*
unfolding *P-Gauss-Jordan-def*
unfolding *Let-def Gauss-Jordan-PA-eq* **by** *presburger*

end

11 Bases of the four fundamental subspaces

theory *Bases-Of-Fundamental-Subspaces*
imports
Gauss-Jordan-PA
begin

11.1 Computation of the bases of the fundamental subspaces

definition *basis-null-space A* = {*row i* (*P-Gauss-Jordan* (*transpose A*)) | *i. to-nat i* ≥ *rank A*}
definition *basis-row-space A* = {*row i* (*Gauss-Jordan A*) |*i. row i* (*Gauss-Jordan A*) ≠ *0*}
definition *basis-col-space A* = {*row i* (*Gauss-Jordan* (*transpose A*)) |*i. row i* (*Gauss-Jordan* (*transpose A*)) ≠ *0*}
definition *basis-left-null-space A* = {*row i* (*P-Gauss-Jordan A*) | *i. to-nat i* ≥ *rank A*}

11.2 Relatioships amongst the bases

lemma *basis-null-space-eq-basis-left-null-space-transpose*:
basis-null-space A = *basis-left-null-space* (*transpose A*)
unfolding *basis-null-space-def*
unfolding *basis-left-null-space-def*
unfolding *rank-transpose*[*of A*, *symmetric*] **..**

lemma *basis-null-space-transpose-eq-basis-left-null-space*:
shows *basis-null-space* (*transpose A*) = *basis-left-null-space A*

by (*metis transpose-transpose basis-null-space-eq-basis-left-null-space-transpose*)

lemma *basis-col-space-eq-basis-row-space-transpose*:
basis-col-space A = *basis-row-space* (*transpose A*)
unfolding *basis-col-space-def basis-row-space-def* **..**

11.3 Code equations

Code equations to make more efficient the computations.

lemma *basis-null-space-code*[*code*]: *basis-null-space A* = (*let GJ* = *Gauss-Jordan-PA* (*transpose A*);
rank-A = (*if A* = *0 then 0 else to-nat* (*GREATEST a. row a* (*snd GJ*) ≠ *0*) + *1*)
in {*row i* (*fst GJ*) | *i. to-nat i* ≥ *rank-A*})
unfolding *basis-null-space-def Let-def P-Gauss-Jordan-def*
unfolding *Gauss-Jordan-PA-eq*
unfolding *rank-transpose*[*symmetric*, *of A*]
unfolding *rank-Gauss-Jordan-code*[*of transpose A*]
unfolding *Let-def*
unfolding *transpose-zero* **..**

lemma *basis-row-space-code*[*code*]: *basis-row-space A* = (*let A'* = *Gauss-Jordan A in* {*row i A'* |*i. row i A'* ≠ *0*})
unfolding *basis-row-space-def Let-def* **..**

lemma *basis-col-space-code*[*code*]: *basis-col-space A* = (*let A'* = *Gauss-Jordan* (*transpose A*) *in* {*row i A'* |*i. row i A'* ≠ *0*})
unfolding *basis-col-space-def Let-def* **..**

lemma *basis-left-null-space-code*[*code*]: *basis-left-null-space A* = (*let GJ* = *Gauss-Jordan-PA A*;
rank-A = (*if A* = *0 then 0 else to-nat* (*GREATEST a. row a* (*snd GJ*) ≠ *0*) + *1*)
in {*row i* (*fst GJ*) | *i. to-nat i* ≥ *rank-A*})
unfolding *basis-left-null-space-def Let-def P-Gauss-Jordan-def*
unfolding *Gauss-Jordan-PA-eq*
unfolding *rank-Gauss-Jordan-code*[*of A*]
unfolding *Let-def*
unfolding *transpose-zero* **..**

11.4 Demonstrations that they are bases

We prove that we have obtained a basis for each subspace

lemma *independent-basis-left-null-space*:
fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
shows *vec.independent* (*basis-left-null-space A*)

proof (*unfold basis-left-null-space-def*, *rule vec.independent-mono*)
show *vec.independent* (*rows* (*P-Gauss-Jordan A*))
by (*metis P-Gauss-Jordan-def det-dependent-rows invertible-det-nz invertible-fst-Gauss-Jordan-PA*)
show {*row i* (*P-Gauss-Jordan A*) |*i*. *rank A* ≤ *to-nat i*} ⊆ (*rows* (*P-Gauss-Jordan A*)) **unfolding** *rows-def* **by** *fast*
qed

lemma *card-basis-left-null-space-eq-dim*:
fixes *A*::'*a*::{*field*}^'*cols*::{*mod-type*}^'*rows*::{*mod-type*}
shows *card* (*basis-left-null-space A*) = *vec.dim* (*left-null-space A*)
proof −
let *?f*=λ*n*. *row* (*from-nat* (*n* + (*rank A*))) (*P-Gauss-Jordan A*)
have *card* (*basis-left-null-space A*) = *card* {*row i* (*P-Gauss-Jordan A*) | *i*. *to-nat i* ≥ *rank A*} **unfolding** *basis-left-null-space-def* **..**
also have ... = *card* {..<(*vec.dimension TYPE*('*a*) *TYPE*('*rows*)) − *rank A*}
proof (*rule bij-betw-same-card*[*symmetric*, *of ?f*], *unfold bij-betw-def*, *rule conjI*)
show *inj-on ?f* {..<(*vec.dimension TYPE*('*a*) *TYPE*('*rows*)) − *rank A*} **unfolding** *inj-on-def*
proof (*auto*, *rule ccontr*, *unfold dimension-vector*)
fix *x y*
assume *x*: *x* <*CARD*('*rows*) − *rank A*
and *y*: *y* < *CARD*('*rows*) − *rank A*
and *eq*: *row* (*from-nat* (*x* + *rank A*)) (*P-Gauss-Jordan A*) = *row* (*from-nat* (*y* + *rank A*)) (*P-Gauss-Jordan A*)
and *x-not-y*: *x* ≠ *y*
have *det* (*P-Gauss-Jordan A*) = *0*
proof (*rule det-identical-rows*[*OF* - *eq*])
have (*x* + *rank A*) ≠ (*y* + *rank A*) **using** *x-not-y x y* **by** *simp*
thus (*from-nat* (*x* + *rank A*)::'*rows*) ≠ *from-nat* (*y* + *rank A*) **by** (*metis* (*mono-tags*) *from-nat-eq-imp-eq less-diff-conv x y*)
qed
moreover have *invertible* (*P-Gauss-Jordan A*) **by** (*metis P-Gauss-Jordan-def invertible-fst-Gauss-Jordan-PA*)
ultimately show *False* **unfolding** *invertible-det-nz* **by** *contradiction*
qed
show *?f* ' {..<(*vec.dimension TYPE*('*a*) *TYPE*('*rows*)) − *rank A*} = {*row i* (*P-Gauss-Jordan A*) |*i*. *rank A* ≤ *to-nat i*}
proof (*unfold image-def dimension-vector*, *auto*, *metis le-add2 less-diff-conv to-nat-from-nat-id*)
fix *i*::'*rows*
assume *rank-le-i*: *rank A* ≤ *to-nat i*
show ∃*x*∈{..<*CARD*('*rows*) − *rank A*}. *row i* (*P-Gauss-Jordan A*) = *row* (*from-nat* (*x* + *rank A*)) (*P-Gauss-Jordan A*)
proof (*rule bexI*[*of* - (*to-nat i* − *rank A*)])
have *i* = (*from-nat* (*to-nat i* − *rank A* + *rank A*)) **by** (*metis rank-le-i from-nat-to-nat-id le-add-diff-inverse2*)
thus *row i* (*P-Gauss-Jordan A*) = *row* (*from-nat* (*to-nat i* − *rank A* + *rank A*)) (*P-Gauss-Jordan A*) **by** *presburger*
show *to-nat i* − *rank A* ∈ {..<*CARD*('*rows*) − *rank A*} **using** *rank-le-i*

by (*metis diff-less-mono lessThan-def mem-Collect-eq to-nat-less-card*)
qed
qed
qed
also have ... = (*vec.dimension TYPE('a) TYPE('rows)*) − *rank A* **unfolding** *card-lessThan* **..**
also have ... = *vec.dim* (*null-space* (*transpose A*)) **unfolding** *dim-null-space rank-transpose* **..**
also have ... = *vec.dim* (*left-null-space A*) **unfolding** *left-null-space-eq-null-space-transpose* **..**
finally show *?thesis* **.**
qed

lemma *basis-left-null-space-in-left-null-space*:
fixes *A*::'*a*::{*field*}^'*cols*::{*mod-type*}^'*rows*::{*mod-type*}
shows *basis-left-null-space A* ⊆ *left-null-space A*
proof (*unfold basis-left-null-space-def left-null-space-def*, *auto*)
fix *i*::'*rows*
assume *rank-le-i*: *rank A* ≤ *to-nat i*
have *row i* (*P-Gauss-Jordan A*) *v*∗ *A* = ((*P-Gauss-Jordan A*) $ *i*) *v*∗ *A* **unfolding** *row-def vec-nth-inverse* **..**
also have ... = ((*P-Gauss-Jordan A*) ∗∗ *A*) $ *i* **unfolding** *row-matrix-matrix-mult* **by** *simp*
also have ... = (*Gauss-Jordan A*) $ *i* **unfolding** *P-Gauss-Jordan-def Gauss-Jordan-PA-eq*[*symmetric*] **using** *fst-Gauss-Jordan-PA* **by** *metis*
also have ... = *0* **by** (*rule rank-less-row-i-imp-i-is-zero*[*OF rank-le-i*])
finally show *row i* (*P-Gauss-Jordan A*) *v*∗ *A* = *0* **.**
qed

lemma *left-null-space-subset-span-basis*:
fixes *A*::'*a*::{*field*}^'*cols*::{*mod-type*}^'*rows*::{*mod-type*}
shows *left-null-space A* ⊆ *vec.span* (*basis-left-null-space A*)
proof (*rule vec.card-ge-dim-independent*)
show *basis-left-null-space A* ⊆ *left-null-space A* **by** (*rule basis-left-null-space-in-left-null-space*)
show *vec.independent* (*basis-left-null-space A*) **by** (*rule independent-basis-left-null-space*)
show *vec.dim* (*left-null-space A*) ≤ *card* (*basis-left-null-space A*)
proof −
have {*x*. *x v*∗ *A* = *0*} = {*x*. (*transpose A*) ∗*v x* = *0*} **by** (*metis transpose-vector*)
thus *?thesis* **using** *card-basis-left-null-space-eq-dim* **by** (*metis order-refl*)
qed
qed

corollary *basis-left-null-space*:
fixes *A*::'*a*::{*field*}^'*cols*::{*mod-type*}^'*rows*::{*mod-type*}
shows *vec.independent* (*basis-left-null-space A*) ∧
left-null-space A = *vec.span* (*basis-left-null-space A*)
by (*metis basis-left-null-space-in-left-null-space independent-basis-left-null-space*

left-null-space-subset-span-basis vec.span-subspace subspace-left-null-space)

corollary *basis-null-space*:
fixes *A*::*'a*::{*field*}^*cols*::{*mod-type*}^*rows*::{*mod-type*}
shows *vec.independent* (*basis-null-space A*) ∧
null-space A = *vec.span* (*basis-null-space A*)
unfolding *basis-null-space-eq-basis-left-null-space-transpose*
unfolding *null-space-eq-left-null-space-transpose*
by (*rule basis-left-null-space*)

lemma *basis-row-space-subset-row-space*:
fixes *A*::*'a*::{*field*}^*cols*::{*mod-type*}^*rows*::{*mod-type*}
shows *basis-row-space A* ⊆ *row-space A*
proof −
have *basis-row-space A* = {*row i* (*Gauss-Jordan A*) |*i*. *row i* (*Gauss-Jordan A*) ≠ *0*} **unfolding** *basis-row-space-def* **..**
also have ... ⊆ *row-space* (*Gauss-Jordan A*)
proof (*unfold row-space-def*, *clarify*)
fix *i* **assume** *row i* (*Gauss-Jordan A*) ≠ *0*
show *row i* (*Gauss-Jordan A*) ∈ *vec.span* (*rows* (*Gauss-Jordan A*))
using *rows-def vec.span-base* **by** *auto*
qed
also have ... = *row-space A* **unfolding** *Gauss-Jordan-PA-eq*[*symmetric*] **unfolding** *fst-Gauss-Jordan-PA*[*symmetric*]
by (*rule row-space-is-preserved*[*OF invertible-fst-Gauss-Jordan-PA*])
finally show *?thesis* **.**
qed

lemma *row-space-subset-span-basis-row-space*:
fixes *A*::*'a*::{*field*}^*cols*::{*mod-type*}^*rows*::{*mod-type*}
shows *row-space A* ⊆ *vec.span* (*basis-row-space A*)
proof (*rule vec.card-ge-dim-independent*)
show *basis-row-space A* ⊆ *row-space A* **by** (*rule basis-row-space-subset-row-space*)
show *vec.independent* (*basis-row-space A*) **unfolding** *basis-row-space-def* **by** (*rule independent-not-zero-rows-rref*[*OF rref-Gauss-Jordan*])
show *vec.dim* (*row-space A*) ≤ *card* (*basis-row-space A*)
unfolding *basis-row-space-def*
using *rref-rank*[*OF rref-Gauss-Jordan*, *of A*] **unfolding** *row-rank-def*[*symmetric*] *rank-def*[*symmetric*] *rank-Gauss-Jordan*[*symmetric*] **by** *fastforce*
qed

lemma *basis-row-space*:
fixes *A*::*'a*::{*field*}^*cols*::{*mod-type*}^*rows*::{*mod-type*}
shows *vec.independent* (*basis-row-space A*)

∧ *vec.span* (*basis-row-space A*) = *row-space A*
proof (*rule conjI*)
show *vec.independent* (*basis-row-space A*) **unfolding** *basis-row-space-def* **using** *independent-not-zero-rows-rref*[*OF rref-Gauss-Jordan*] **.**
show *vec.span* (*basis-row-space A*) = *row-space A*
proof (*rule vec.span-subspace*)
show *basis-row-space A* ⊆ *row-space A* **by** (*rule basis-row-space-subset-row-space*)
show *row-space A* ⊆ *vec.span* (*basis-row-space A*) **by** (*rule row-space-subset-span-basis-row-space*)
show *vec.subspace* (*row-space A*) **by** (*rule subspace-row-space*)
qed
qed

corollary *basis-col-space*:
fixes *A*::'*a*::{*field*}^'*cols*::{*mod-type*}^'*rows*::{*mod-type*}
shows *vec.independent* (*basis-col-space A*)
∧ *vec.span* (*basis-col-space A*) = *col-space A*
unfolding *col-space-eq-row-space-transpose basis-col-space-eq-basis-row-space-transpose*
by (*rule basis-row-space*)

end

12 Solving systems of equations using the Gauss Jordan algorithm

theory *System-Of-Equations*
imports
Gauss-Jordan-PA
Bases-Of-Fundamental-Subspaces
begin

12.1 Definitions

Given a system of equations $A *v x = b$, the following function returns the pair ($P ** A$, $P *v b$), where P is the matrix which states *Gauss-Jordan A* = $P ** A$. That matrix is computed by means of *Gauss-Jordan-PA*.

definition *solve-system* :: ('*a*::{*field*}^'*cols*::{*mod-type*}^'*rows*::{*mod-type*}) ⇒ ('*a*^'*rows*::{*mod-type*})

⇒ (('*a*^'*cols*::{*mod-type*}^'*rows*::{*mod-type*}) × ('*a*^'*rows*::{*mod-type*}))
where *solve-system A b* = (*let A'* = *Gauss-Jordan-PA A in* (*snd A'*, (*fst A'*) *∗v b*))

definition *is-solution* **where** *is-solution x A b* = (*A ∗v x* = *b*)

12.2 Relationship between *is-solution-def* and *solve-system-def*

lemma *is-solution-imp-solve-system*:
fixes *A*::'*a*::{*field*}^'*cols*::{*mod-type*}^'*rows*::{*mod-type*}
assumes *xAb*:*is-solution x A b*

shows *is-solution x* (*fst* (*solve-system A b*)) (*snd* (*solve-system A b*))
proof −
have (*fst* (*Gauss-Jordan-PA A*)∗*v*(*A* ∗*v x*) = *fst* (*Gauss-Jordan-PA A*) ∗*v b*)
using *xAb* **unfolding** *is-solution-def* **by** *fast*
hence (*snd* (*Gauss-Jordan-PA A*) ∗*v x* = *fst* (*Gauss-Jordan-PA A*) ∗*v b*)
unfolding *matrix-vector-mul-assoc*
unfolding *fst-Gauss-Jordan-PA*[*of A*] .
thus *is-solution x* (*fst* (*solve-system A b*)) (*snd* (*solve-system A b*))
unfolding *is-solution-def solve-system-def Let-def* **by** *simp*
qed

lemma *solve-system-imp-is-solution*:
fixes *A*::*'a*::{*field*}^*cols*::{*mod-type*}^*rows*::{*mod-type*}
assumes *xAb*: *is-solution x* (*fst* (*solve-system A b*)) (*snd* (*solve-system A b*))
shows *is-solution x A b*
proof −
have *fst* (*solve-system A b*) ∗*v x* = *snd* (*solve-system A b*)
using *xAb* **unfolding** *is-solution-def* .
hence *snd* (*Gauss-Jordan-PA A*) ∗*v x* = *fst* (*Gauss-Jordan-PA A*) ∗*v b*
unfolding *solve-system-def Let-def fst-conv snd-conv* .
hence (*fst* (*Gauss-Jordan-PA A*) ∗∗ *A*) ∗*v x* = *fst* (*Gauss-Jordan-PA A*) ∗*v b*
unfolding *fst-Gauss-Jordan-PA* .
hence *fst* (*Gauss-Jordan-PA A*) ∗*v* (*A* ∗*v x*) = *fst* (*Gauss-Jordan-PA A*) ∗*v b*
unfolding *matrix-vector-mul-assoc* .
hence *matrix-inv* (*fst* (*Gauss-Jordan-PA A*)) ∗*v* (*fst* (*Gauss-Jordan-PA A*) ∗*v* (*A* ∗*v x*))
= *matrix-inv* (*fst* (*Gauss-Jordan-PA A*)) ∗*v* (*fst* (*Gauss-Jordan-PA A*) ∗*v b*) **by** *simp*
hence (*A* ∗*v x*) = *b*
unfolding *matrix-vector-mul-assoc*[*of matrix-inv* (*fst* (*Gauss-Jordan-PA A*))]
unfolding *matrix-inv-left*[*OF invertible-fst-Gauss-Jordan-PA*]
unfolding *matrix-vector-mul-lid* .
thus *?thesis* **unfolding** *is-solution-def* .
qed

lemma *is-solution-solve-system*:
fixes *A*::*'a*::{*field*}^*cols*::{*mod-type*}^*rows*::{*mod-type*}
shows *is-solution x A b* = *is-solution x* (*fst* (*solve-system A b*)) (*snd* (*solve-system A b*))
using *solve-system-imp-is-solution is-solution-imp-solve-system* **by** *blast*

12.3 Consistent and inconsistent systems of equations

definition *consistent* :: *'a*::{*field*}^*cols*::{*mod-type*}^*rows*::{*mod-type*} ⇒ *'a*::{*field*}^*rows*::{*mod-type*} ⇒ *bool*
where *consistent A b* = (∃ *x*. *is-solution x A b*)

definition *inconsistent* **where** *inconsistent A b* = (¬ (*consistent A b*))

lemma *inconsistent*: *inconsistent A b* = (¬ (∃ *x*. *is-solution x A b*))
unfolding *inconsistent-def consistent-def* **by** *simp*

The following function will be use to solve consistent systems which are already in the reduced row echelon form.

definition *solve-consistent-rref* :: *'a*::{*field*}^'*cols*::{*mod-type*}^'*rows*::{*mod-type*} ⇒ *'a*::{*field*}^'*rows*::{*mod-type*} ⇒ *'a*::{*field*}^'*cols*::{*mod-type*}
where *solve-consistent-rref A b* = (χ *j*. *if* (∃ *i*. *A* $ *i* $ *j* = *1* ∧ *j*=(*LEAST n*. *A* $ *i* $ *n* ≠ *0*)) *then b* $ (*THE i*. *A* $ *i* $ *j* = *1*) *else 0*)

lemma *solve-consistent-rref-code*[*code abstract*]:
shows *vec-nth* (*solve-consistent-rref A b*) = (% *j*. *if* (∃ *i*. *A* $ *i* $ *j* = *1* ∧ *j*=(*LEAST n*. *A* $ *i* $ *n* ≠ *0*)) *then b* $ (*THE i*. *A* $ *i* $ *j* = *1*) *else 0*)
unfolding *solve-consistent-rref-def* **by** *auto*

lemma *rank-ge-imp-is-solution*:
fixes *A*::*'a*::{*field*}^'*cols*::{*mod-type*}^'*rows*::{*mod-type*}
assumes *con*: *rank A* ≥ (*if* (∃ *a*. (*P-Gauss-Jordan A* ∗*v b*) $ *a* ≠ *0*)
then (*to-nat* (*GREATEST a*. (*P-Gauss-Jordan A* ∗*v b*) $ *a* ≠ *0*) + *1*) *else 0*)
shows *is-solution* (*solve-consistent-rref* (*Gauss-Jordan A*) (*P-Gauss-Jordan A* ∗*v b*)) *A b*
proof −
have *is-solution* (*solve-consistent-rref* (*Gauss-Jordan A*) (*P-Gauss-Jordan A* ∗*v b*)) (*Gauss-Jordan A*) (*P-Gauss-Jordan A* ∗*v b*)
proof (*unfold is-solution-def solve-consistent-rref-def*, *subst matrix-vector-mult-def*, *vector*, *auto*)
fix *a*
let *?f*=λ*j*. *Gauss-Jordan A* $ *a* $ *j* ∗
(*if* ∃ *i*. *Gauss-Jordan A* $ *i* $ *j* = *1* ∧ *j* = (*LEAST n*. *Gauss-Jordan A* $ *i* $ *n* ≠ *0*)
then (*P-Gauss-Jordan A* ∗*v b*) $ (*THE i*. *Gauss-Jordan A* $ *i* $ *j* = *1*) *else 0*)
show *sum ?f UNIV* = (*P-Gauss-Jordan A* ∗*v b*) $ *a*
proof (*cases A*=*0*)
case *True*
hence *rank-A-eq-0*:*rank A* = *0* **using** *rank-0* **by** *simp*
have (*P-Gauss-Jordan A* ∗*v b*) = *0*
proof (*rule ccontr*)
assume *not-zero*: *P-Gauss-Jordan A* ∗*v b* ≠ *0*
hence *ex-a*: ∃ *a*. (*P-Gauss-Jordan A* ∗*v b*) $ *a* ≠ *0* **by** (*metis vec-eq-iff zero-index*)
show *False* **using** *con* **unfolding** *if-P*[*OF ex-a*] **unfolding** *rank-A-eq-0* **by** *auto*
qed

thus *?thesis* **unfolding** *A-0-imp-Gauss-Jordan-0*[*OF True*] **by** *force*

next
case *False* **note** *A-not-zero=False*
define *not-zero-positions-row-a* **where** *not-zero-positions-row-a* = {*j*. *Gauss-Jordan A* $ *a* $ *j* ≠ *0*}
define *zero-positions-row-a* **where** *zero-positions-row-a* = {*j*. *Gauss-Jordan A* $ *a* $ *j* = *0*}
have *UNIV-rw*: *UNIV* = *not-zero-positions-row-a* ∪ *zero-positions-row-a*
unfolding *zero-positions-row-a-def not-zero-positions-row-a-def* **by** *auto*
have *disj*: *not-zero-positions-row-a* ∩ *zero-positions-row-a* = {}
unfolding *zero-positions-row-a-def not-zero-positions-row-a-def* **by** *fastforce*
have *sum-zero*: (*sum ?f zero-positions-row-a*) = *0*
by (*unfold zero-positions-row-a-def*, *rule sum.neutral*, *fastforce*)
have *sum ?f* (*UNIV*::*'cols set*)=*sum ?f* (*not-zero-positions-row-a* ∪ *zero-positions-row-a*)

unfolding *UNIV-rw* **..**
also have ... = *sum ?f* (*not-zero-positions-row-a*) + (*sum ?f zero-positions-row-a*)

by (*rule sum.union-disjoint*[*OF - - disj*], *simp+*)
also have ... = *sum ?f* (*not-zero-positions-row-a*) **unfolding** *sum-zero* **by** *simp*
also have ... = (*P-Gauss-Jordan A* ∗*v b*) $ *a*
proof (*cases not-zero-positions-row-a* = {})
case *True* **note** *zero-row-a=True*
show *?thesis*
proof (*cases* ∃ *a*. (*P-Gauss-Jordan A* ∗*v b*) $ *a* ≠ *0*)
case *False* **hence** (*P-Gauss-Jordan A* ∗*v b*) $ *a* = *0* **by** *simp*
thus *?thesis* **unfolding** *True* **by** *auto*
next
case *True*
have *rank-not-0*: *rank A* ≠ *0* **by** (*metis A-not-zero less-not-refl3 rank-Gauss-Jordan rank-greater-zero*)
have *greatest-less-a*: (*GREATEST a*. ¬ *is-zero-row a* (*Gauss-Jordan A*)) < *a*
proof (*unfold is-zero-row-def*, *rule greatest-less-zero-row*)
show *reduced-row-echelon-form-upt-k* (*Gauss-Jordan A*) (*ncols* (*Gauss-Jordan A*))
using *rref-Gauss-Jordan* **unfolding** *reduced-row-echelon-form-def* **.**
show *is-zero-row-upt-k a* (*ncols* (*Gauss-Jordan A*)) (*Gauss-Jordan A*)
by (*metis* (*mono-tags*) *Collect-empty-eq is-zero-row-upt-ncols not-zero-positions-row-a-def zero-row-a*)
show ¬ (∀ *a*. *is-zero-row-upt-k a* (*ncols* (*Gauss-Jordan A*)) (*Gauss-Jordan A*))
by (*metis False Gauss-Jordan-not-0 is-zero-row-upt-ncols vec-eq-iff zero-index*)
qed
hence *to-nat* (*GREATEST a*. ¬ *is-zero-row a* (*Gauss-Jordan A*)) < *to-nat a* **using** *to-nat-mono* **by** *fast*
hence *rank-le-to-nat-a*: *rank A* ≤ *to-nat a* **unfolding** *rank-eq-suc-to-nat-greatest*[*OF A-not-zero*] **by** *simp*

have *to-nat* (*GREATEST a*. (*P-Gauss-Jordan A* ∗*v b*) $ *a* ≠ *0*) < *to-nat a*
using *con* **unfolding** *consistent-def* **unfolding** *if-P*[*OF True*] **using** *rank-le-to-nat-a* **by** *simp*
hence (*GREATEST a*. (*P-Gauss-Jordan A* ∗*v b*) $ *a* ≠ *0*) < *a* **by** (*metis not-le to-nat-mono'*)
hence (*P-Gauss-Jordan A* ∗*v b*) $ *a* = *0* **using** *not-greater-Greatest* **by** *blast*
thus *?thesis* **unfolding** *zero-row-a* **by** *simp*
qed
next
case *False* **note** *not-empty=False*
have *not-zero-positions-row-a-rw*: *not-zero-positions-row-a* = {*LEAST j*. (*Gauss-Jordan A*) $ *a* $ *j* ≠ *0*} ∪ (*not-zero-positions-row-a* − {*LEAST j*. (*Gauss-Jordan A*) $ *a* $ *j* ≠ *0*})
unfolding *not-zero-positions-row-a-def*
by (*metis* (*mono-tags*) *Collect-cong False LeastI-ex bot-set-def empty-iff insert-Diff-single insert-absorb insert-is-Un mem-Collect-eq not-zero-positions-row-a-def*)

have *sum-zero'*: *sum ?f* (*not-zero-positions-row-a* − {*LEAST j*. (*Gauss-Jordan A*) $ *a* $ *j* ≠ *0*}) = *0*
by (*rule sum.neutral*, *auto*, *metis is-zero-row-def' rref-Gauss-Jordan rref-condition4-explicit zero-neq-one*)
have *sum ?f* (*not-zero-positions-row-a*) = *sum ?f* {*LEAST j*. (*Gauss-Jordan A*) $ *a* $ *j* ≠ *0*} + *sum ?f* (*not-zero-positions-row-a* − {*LEAST j*. (*Gauss-Jordan A*) $ *a* $ *j* ≠ *0*})
by (*subst not-zero-positions-row-a-rw*, *rule sum.union-disjoint*[*OF - - -*], *simp+*)
also have ... = *?f* (*LEAST j*. (*Gauss-Jordan A*) $ *a* $ *j* ≠ *0*) **using** *sum-zero'* **by** *force*
also have ... = (*P-Gauss-Jordan A* ∗*v b*) $ *a*
proof (*cases* ∃ *i*. (*Gauss-Jordan A*) $ *i* $ (*LEAST j*. (*Gauss-Jordan A*) $ *a* $ *j* ≠ *0*) = *1* ∧ (*LEAST j*. (*Gauss-Jordan A*) $ *a* $ *j* ≠ *0*) = (*LEAST n*. (*Gauss-Jordan A*) $ *i* $ *n* ≠ *0*))
case *True*
have *A-least-eq-1*: (*Gauss-Jordan A*) $ *a* $ (*LEAST j*. (*Gauss-Jordan A*) $ *a* $ *j* ≠ *0*) = *1*
by (*metis* (*mono-tags*) *empty-Collect-eq is-zero-row-def' not-empty not-zero-positions-row-a-def rref-Gauss-Jordan rref-condition2-explicit*)
moreover have (*THE i*. (*Gauss-Jordan A*) $ *i* $ (*LEAST j*. (*Gauss-Jordan A*) $ *a* $ *j* ≠ *0*) = *1*) = *a*
proof (*rule the-equality*)
show (*Gauss-Jordan A*) $ *a* $ (*LEAST j*. (*Gauss-Jordan A*) $ *a* $ *j* ≠ *0*) = *1* **using** *A-least-eq-1* **.**
show ⋀*i*. (*Gauss-Jordan A*) $ *i* $ (*LEAST j*. (*Gauss-Jordan A*) $ *a* $ *j* ≠ *0*) = *1* ⟹ *i* = *a*
by (*metis calculation is-zero-row-def' rref-Gauss-Jordan rref-condition4-explicit zero-neq-one*)
qed

ultimately show *?thesis* **unfolding** *if-P*[*OF True*] **by** *simp*
next
case *False*
have *is-zero-row a* (*Gauss-Jordan A*) **using** *False rref-Gauss-Jordan rref-condition2* **by** *blast*
hence (*P-Gauss-Jordan A* ∗*v b*) $ *a* = *0*
by (*metis* (*mono-tags*) *IntI disj empty-iff insert-compr insert-is-Un is-zero-row-def' mem-Collect-eq not-zero-positions-row-a-rw zero-positions-row-a-def*)
thus *?thesis* **unfolding** *if-not-P*[*OF False*] **by** *fastforce*
qed
finally show *?thesis* **.**
qed
finally show *sum ?f UNIV* = (*P-Gauss-Jordan A* ∗*v b*) $ *a* **.**
qed
qed
thus *?thesis* **apply** (*subst is-solution-solve-system*)
unfolding *solve-system-def Let-def snd-conv fst-conv* **unfolding** *Gauss-Jordan-PA-eq P-Gauss-Jordan-def* **.**
qed

corollary *rank-ge-imp-consistent*:
fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *rank A* ≥ (*if* (∃ *a*. (*P-Gauss-Jordan A* ∗*v b*) $ *a* ≠ *0*) *then* (*to-nat* (*GREATEST a*. (*P-Gauss-Jordan A* ∗*v b*) $ *a* ≠ *0*) + *1*) *else 0*)
shows *consistent A b*
using *rank-ge-imp-is-solution assms* **unfolding** *consistent-def* **by** *auto*

lemma *inconsistent-imp-rank-less*:
fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *inc*: *inconsistent A b*
shows *rank A* < (*if* (∃ *a*. (*P-Gauss-Jordan A* ∗*v b*) $ *a* ≠ *0*) *then* (*to-nat* (*GREATEST a*. (*P-Gauss-Jordan A* ∗*v b*) $ *a* ≠ *0*) + *1*) *else 0*)
proof (*rule ccontr*)
assume ¬ *rank A* < (*if* ∃ *a*. (*P-Gauss-Jordan A* ∗*v b*) $ *a* ≠ *0 then to-nat* (*GREATEST a*. (*P-Gauss-Jordan A* ∗*v b*) $ *a* ≠ *0*) + *1 else 0*)
hence (*if* ∃ *a*. (*P-Gauss-Jordan A* ∗*v b*) $ *a* ≠ *0 then to-nat* (*GREATEST a*. (*P-Gauss-Jordan A* ∗*v b*) $ *a* ≠ *0*) + *1 else 0*) ≤ *rank A* **by** *simp*
hence *consistent A b* **using** *rank-ge-imp-consistent* **by** *auto*
thus *False* **using** *inc* **unfolding** *inconsistent-def* **by** *contradiction*
qed

lemma *rank-less-imp-inconsistent*:
fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *inc*: *rank A* < (*if* (∃ *a*. (*P-Gauss-Jordan A* ∗*v b*) $ *a* ≠ *0*) *then* (*to-nat* (*GREATEST a*. (*P-Gauss-Jordan A* ∗*v b*) $ *a* ≠ *0*) + *1*) *else 0*)
shows *inconsistent A b*
proof (*rule ccontr*)

```
  define i where i = (GREATEST a. (P-Gauss-Jordan A *v b) $ a ≠ 0)
  define j where j = (GREATEST a. ¬ is-zero-row a (Gauss-Jordan A))
  assume ¬ inconsistent A b
 hence ex-solution: ∃ x. is-solution x A b unfolding inconsistent-def consistent-def
by auto
  from this obtain x where is-solution x A b  by auto
  hence is-solution-solve: is-solution x (Gauss-Jordan A) (P-Gauss-Jordan A *v
b)
    using is-solution-solve-system
  by (metis Gauss-Jordan-PA-eq P-Gauss-Jordan-def fst-conv snd-conv solve-system-def)
  show False
  proof (cases A=0)
    case True
    hence rank-eq-0: rank A = 0 using rank-0 by simp
    hence exists-not-0:(∃ a. (P-Gauss-Jordan A *v b) $ a ≠ 0)
      using inc unfolding inconsistent
       using to-nat-plus-1-set[of (GREATEST a. (P-Gauss-Jordan A *v b) $ a ≠
0)]
      by presburger
    show False
      using True ‹is-solution x A b› exists-not-0 is-solution-def by force
  next
    case False
    have j-less-i: j<i
    proof −
      have rank-less-greatest-i: rank A < to-nat i + 1
        using inc unfolding i-def inconsistent by presburger
       moreover have rank-eq-greatest-A: rank A = to-nat j + 1 unfolding j-def
by (rule rank-eq-suc-to-nat-greatest[OF False])
      ultimately have to-nat j + 1 < to-nat i + 1 by simp
      hence to-nat j < to-nat i by auto
      thus j<i by (metis (full-types) not-le to-nat-mono')
    qed
   have is-zero-i: is-zero-row i (Gauss-Jordan A) by (metis (full-types) j-def j-less-i
not-greater-Greatest)
    have (Gauss-Jordan A *v x) $ i = 0
    proof (unfold matrix-vector-mult-def, auto, rule sum.neutral,clarify)
      fix a::'cols
        show Gauss-Jordan A $ i $ a * x $ a = 0 using is-zero-i unfolding
is-zero-row-def' by simp
    qed
    moreover have (Gauss-Jordan A *v x) $ i ≠ 0
    proof −
       have Gauss-Jordan A *v x = P-Gauss-Jordan A *v b using is-solution-def
is-solution-solve by blast
      also have ... $ i ≠ 0
        unfolding i-def
      proof (rule GreatestI-ex)
         show ∃ x. (P-Gauss-Jordan A *v b) $ x ≠ 0 using inc unfolding i-def
```

inconsistent **by** *presburger*
    qed
    finally show *?thesis* **.**
   qed
   ultimately show *False* **by** *contradiction*
  qed
qed

corollary *consistent-imp-rank-ge*:
  fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
  assumes *consistent A b*
  shows *rank A* ≥ (*if* (∃ *a*. (*P-Gauss-Jordan A* ∗*v b*) $ *a* ≠ *0*) *then* (*to-nat* (*GREATEST a*. (*P-Gauss-Jordan A* ∗*v b*) $ *a* ≠ *0*) + *1*) *else 0*)
  using *rank-less-imp-inconsistent* **by** (*metis assms inconsistent-def not-less*)

lemma *inconsistent-eq-rank-less*:
  fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
  shows *inconsistent A b* = (*rank A* < (*if* (∃ *a*. (*P-Gauss-Jordan A* ∗*v b*) $ *a* ≠ *0*)
                then (*to-nat* (*GREATEST a*. (*P-Gauss-Jordan A* ∗*v b*) $ *a* ≠ *0*) + *1*) *else 0*))
  using *inconsistent-imp-rank-less rank-less-imp-inconsistent* **by** *blast*

lemma *consistent-eq-rank-ge*:
  fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
  shows *consistent A b* = (*rank A* ≥ (*if* (∃ *a*. (*P-Gauss-Jordan A* ∗*v b*) $ *a* ≠ *0*)
                then (*to-nat* (*GREATEST a*. (*P-Gauss-Jordan A* ∗*v b*) $ *a* ≠ *0*) + *1*) *else 0*))
  using *consistent-imp-rank-ge rank-ge-imp-consistent* **by** *blast*

corollary *consistent-imp-is-solution*:
  fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
  assumes *consistent A b*
  shows *is-solution* (*solve-consistent-rref* (*Gauss-Jordan A*) (*P-Gauss-Jordan A* ∗*v b*)) *A b*
  by (*rule rank-ge-imp-is-solution*[*OF assms*[*unfolded consistent-eq-rank-ge*]])

corollary *consistent-imp-is-solution'*:
  fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
  assumes *consistent A b*
  shows *is-solution* (*solve-consistent-rref* (*fst* (*solve-system A b*)) (*snd* (*solve-system A b*))) *A b*
  using *consistent-imp-is-solution*[*OF assms*] **unfolding** *solve-system-def Let-def snd-conv fst-conv*
  unfolding *Gauss-Jordan-PA-eq P-Gauss-Jordan-def* **.**

Code equations optimized using Lets

lemma *inconsistent-eq-rank-less-code*[*code*]:
fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
shows *inconsistent A b* = (*let GJ-P*=*Gauss-Jordan-PA A*;
P-mult-b = (*fst*(*GJ-P*) ∗*v b*);
rank-A = (*if A* = *0 then 0 else to-nat* (*GREATEST a*. *row a* (*snd GJ-P*) ≠ *0*) + *1*) *in* (*rank-A* < (*if* (∃ *a*. *P-mult-b* $ *a* ≠ *0*)
then (*to-nat* (*GREATEST a*. *P-mult-b* $ *a* ≠ *0*) + *1*) *else 0*)))
unfolding *inconsistent-eq-rank-less Let-def rank-Gauss-Jordan-code*
unfolding *Gauss-Jordan-PA-eq P-Gauss-Jordan-def* **..**

lemma *consistent-eq-rank-ge-code*[*code*]:
fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
shows *consistent A b* = (*let GJ-P*=*Gauss-Jordan-PA A*;
P-mult-b = (*fst*(*GJ-P*) ∗*v b*);
rank-A = (*if A* = *0 then 0 else to-nat* (*GREATEST a*. *row a* (*snd GJ-P*) ≠ *0*) + *1*) *in* (*rank-A* ≥ (*if* (∃ *a*. *P-mult-b* $ *a* ≠ *0*)
then (*to-nat* (*GREATEST a*. *P-mult-b* $ *a* ≠ *0*) + *1*) *else 0*)))
unfolding *consistent-eq-rank-ge Let-def rank-Gauss-Jordan-code*
unfolding *Gauss-Jordan-PA-eq P-Gauss-Jordan-def* **..**

12.4 Solution set of a system of equations. Dependent and independent systems.

definition *solution-set* **where** *solution-set A b* = {*x*. *is-solution x A b*}

lemma *null-space-eq-solution-set*:
shows *null-space A* = *solution-set A 0* **unfolding** *null-space-def solution-set-def is-solution-def* **..**

corollary *dim-solution-set-homogeneous-eq-dim-null-space*[*code-unfold*]:
shows *vec.dim* (*solution-set A 0*) = *vec.dim* (*null-space A*) **using** *null-space-eq-solution-set*[*of A*] **by** *simp*

lemma *zero-is-solution-homogeneous-system*:
shows *0* ∈ (*solution-set A 0*)
unfolding *solution-set-def is-solution-def*
using *matrix-vector-mult-0-right* **by** *fast*

lemma *homogeneous-solution-set-subspace*:
fixes *A*::*'a*::{*field*}^*'n*^*'rows*
shows *vec.subspace* (*solution-set A 0*)
using *subspace-null-space*[*of A*] **unfolding** *null-space-eq-solution-set* **.**

lemma *solution-set-rel*:
fixes *A*::*'a*::{*field*}^*'n*^*'rows*

assumes *p*: *is-solution p A b*
shows *solution-set A b* = {*p*} + (*solution-set A 0*)
proof (*unfold set-plus-def*, *auto*)
fix *ba*
assume *ba*: *ba* ∈ *solution-set A 0*
have *A* ∗*v* (*p* + *ba*) = (*A* ∗*v p*) + (*A* ∗*v ba*) **unfolding** *matrix-vector-right-distrib*
..
also have ... = *b* **using** *p ba* **unfolding** *solution-set-def is-solution-def* **by** *simp*
finally show *p* + *ba* ∈ *solution-set A b* **unfolding** *solution-set-def is-solution-def*
by *simp*
next
fix *x*
assume *x*: *x* ∈ *solution-set A b*
show ∃ *b*∈*solution-set A 0*. *x* = *p* + *b*
proof (*rule bexI*[*of* - *x*−*p*], *simp*)
have *A* ∗*v* (*x* − *p*) = (*A* ∗*v x*) − (*A* ∗*v p*) **by** (*metis* (*no-types*) *add-diff-cancel diff-add-cancel matrix-vector-right-distrib*)
also have ... = *0* **using** *x p* **unfolding** *solution-set-def is-solution-def* **by** *simp*
finally show *x* − *p* ∈ *solution-set A 0* **unfolding** *solution-set-def is-solution-def*
by *simp*
qed
qed

lemma *independent-and-consistent-imp-uniqueness-solution*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *dim-0*: *vec.dim* (*solution-set A 0*) = *0*
and *con*: *consistent A b*
shows ∃!*x*. *is-solution x A b*
proof −
obtain *p* **where** *p*: *is-solution p A b* **using** *con* **unfolding** *consistent-def* **by** *blast*
have *solution-set-0*: *solution-set A 0* = {*0*}
using *vec.dim-zero-eq*[*OF dim-0*] *zero-is-solution-homogeneous-system* **by** *blast*
show ∃!*x*. *is-solution x A b*
proof (*rule ex-ex1I*)
show ∃ *x*. *is-solution x A b* **using** *p* **by** *auto*
fix *x y* **assume** *x*: *is-solution x A b* **and** *y*: *is-solution y A b*
have *solution-set A b* = {*p*} + (*solution-set A 0*) **unfolding** *solution-set-rel*[*OF p*] **..**
also have ... = {*p*} **unfolding** *solution-set-0 set-plus-def* **by** *force*
finally show *x* = *y* **using** *x y* **unfolding** *solution-set-def* **by** (*metis* (*full-types*) *mem-Collect-eq singleton-iff*)
qed
qed

lemma *card-1-exists*: *card s* = *1* ⟷ (∃!*x*. *x* ∈ *s*)
unfolding *One-nat-def*
apply *rule* **apply**(*drule card-eq-SucD*) **defer apply**(*erule ex1E*)

proof−
fix x **assume** as:$x \in s \; \forall y.\; y \in s \longrightarrow y = x$
have $*$:$s = insert\ x\ \{\}$ **apply** − **apply**($rule$,$rule$) **unfolding** *singleton-iff*
apply($rule\ as(2)[rule\text{-}format]$) **using** $as(1)$ **by** *auto*
show $card\ s = Suc\ 0$ **unfolding** $*$ **using** *card.insert-remove* **by** *auto*
qed *auto*

corollary *independent-and-consistent-imp-card-1*:
fixes A::$'a$::{*field*}^$'n$::{*mod-type*}^$'rows$::{*mod-type*}
assumes *dim-0*: *vec.dim* $(solution\text{-}set\ A\ 0) = 0$
and *con*: *consistent* $A\ b$
shows $card\ (solution\text{-}set\ A\ b) = 1$
using *independent-and-consistent-imp-uniqueness-solution*[*OF assms*] **unfolding** *solution-set-def*
using *card-1-exists* **by** *auto*

lemma *uniqueness-solution-imp-independent*:
fixes A::$'a$::{*field*}^$'n$^$'rows$
assumes *ex1-sol*: $\exists!x.\ is\text{-}solution\ x\ A\ b$
shows $vec.dim\ (solution\text{-}set\ A\ 0) = 0$
proof −
obtain x **where** x: *is-solution* $x\ A\ b$ **using** *ex1-sol* **by** *blast*
have *solution-set-homogeneous-zero*: $solution\text{-}set\ A\ 0 = \{0\}$
proof (*rule ccontr*)
assume *not-zero-set*: $solution\text{-}set\ A\ 0 \neq \{0\}$
have *homogeneous-not-empty*: $solution\text{-}set\ A\ 0 \neq \{\}$ **by** (*metis empty-iff zero-is-solution-homogeneous-system*)
obtain y **where** y: $y \in solution\text{-}set\ A\ 0$ **and** *y-not-0*: $y \neq 0$ **using** *not-zero-set homogeneous-not-empty* **by** *blast*
have $\{x\} = solution\text{-}set\ A\ b$ **unfolding** *solution-set-def* **using** x *ex1-sol* **by** *blast*
also have $... = \{x\} + solution\text{-}set\ A\ 0$ **unfolding** *solution-set-rel*[*OF x*] **..**
finally show *False*
by (*metis* (*opaque-lifting*, *mono-tags*) *add-left-cancel monoid-add-class.add.right-neutral empty-iff insert-iff set-plus-intro y y-not-0*)
qed
thus *?thesis* **using** *vec.dim-zero-eq'* **by** *blast*
qed

corollary *uniqueness-solution-eq-independent-and-consistent*:
fixes A::$'a$::{*field*}^$'n$::{*mod-type*}^$'rows$::{*mod-type*}
shows $(\exists!x.\ is\text{-}solution\ x\ A\ b) = (consistent\ A\ b \wedge vec.dim\ (solution\text{-}set\ A\ 0) = 0)$
using *independent-and-consistent-imp-uniqueness-solution uniqueness-solution-imp-independent consistent-def*
by *metis*

lemma *consistent-homogeneous*:
shows *consistent A 0* **unfolding** *consistent-def is-solution-def* **using** *matrix-vector-mult-0-right*
by *fast*

lemma *dim-solution-set-0*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'rows*::{*mod-type*}
shows (*vec.dim* (*solution-set A 0*) = *0*) = (*solution-set A 0* = {*0*})
using *homogeneous-solution-set-subspace vec.dim-zero-subspace-eq* **by** *auto*

We have to impose the restriction *semiring-char-0* in the following lemma, because it may not hold over a general field (for instance, in Z2 there is a finite number of elements, so the solution set can't be infinite.

lemma *dim-solution-set-not-zero-imp-infinite-solutions-homogeneous*:
fixes *A*::*'a*::{*field*, *semiring-char-0*}^*'n*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *dim-not-zero*: *vec.dim* (*solution-set A 0*) > *0*
shows *infinite* (*solution-set A 0*)
proof −
have *solution-set A 0* ≠ {*0*} **using** *vec.dim-zero-subspace-eq*[*of solution-set A 0*]
dim-not-zero
by (*metis less-numeral-extra*(*3*) *vec.dim-zero-eq'*)
from *this* **obtain** *x* **where** *x*: *x* ∈ *solution-set A 0* **and** *x-not-0*: *x* ≠ *0* **using**
vec.subspace-0[*OF homogeneous-solution-set-subspace*, *of A*] **by** *auto*
define *f* **where** *f* = (λ*n*::*nat*. (*of-nat n*) ∗*s x*)
show *?thesis*
proof (*unfold infinite-iff-countable-subset*, *rule exI*[*of - f*], *rule conjI*)
show *inj f* **unfolding** *inj-on-def* **unfolding** *f-def* **using** *x-not-0*
by (*auto simp*: *vec-eq-iff*)
show *range f* ⊆ *solution-set A 0* **using** *homogeneous-solution-set-subspace*
using *x* **unfolding** *vec.subspace-def image-def f-def* **by** *fast*
qed
qed

lemma *infinite-solutions-homogeneous-imp-dim-solution-set-not-zero*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *i*: *infinite* (*solution-set A 0*)
shows *vec.dim* (*solution-set A 0*) > *0*
by (*metis dim-solution-set-0 finite.simps gr0I i*)

corollary *infinite-solution-set-homogeneous-eq*:
fixes *A*::*'a*::{*field*,*semiring-char-0*}^*'n*::{*mod-type*}^*'rows*::{*mod-type*}
shows *infinite* (*solution-set A 0*) = (*vec.dim* (*solution-set A 0*) > *0*)
using *infinite-solutions-homogeneous-imp-dim-solution-set-not-zero*
using *dim-solution-set-not-zero-imp-infinite-solutions-homogeneous* **by** *metis*

corollary *infinite-solution-set-homogeneous-eq'*:
fixes *A*::*'a*::{*field*,*semiring-char-0*}^*'n*::{*mod-type*}^*'rows*::{*mod-type*}
shows (∃$_\infty$*x*. *is-solution x A 0*) = (*vec.dim* (*solution-set A 0*) > *0*)

unfolding *infinite-solution-set-homogeneous-eq*[*symmetric*] *INFM-iff-infinite* **unfolding** *solution-set-def* **..**

lemma *infinite-solution-set-imp-consistent*:
infinite (*solution-set A b*) ⟹ *consistent A b*
by (*auto dest*!: *infinite-imp-nonempty simp*: *solution-set-def consistent-def*)

lemma *dim-solution-set-not-zero-imp-infinite-solutions-no-homogeneous*:
fixes *A*::*'a*::{*field*, *semiring-char-0*}^*n*::{*mod-type*}^*rows*::{*mod-type*}
assumes *dim-not-0*: *vec.dim* (*solution-set A 0*) > *0*
and *con*: *consistent A b*
shows *infinite* (*solution-set A b*)
proof −
have *solution-set A 0* ≠ {*0*} **using** *vec.dim-zero-subspace-eq*[*of solution-set A 0*] *dim-not-0*
by (*metis less-numeral-extra*(*3*) *vec.dim-zero-eq'*)
from *this* **obtain** *x* **where** *x*: *x* ∈ *solution-set A 0* **and** *x-not-0*: *x* ≠ *0* **using** *vec.subspace-0*[*OF homogeneous-solution-set-subspace*, *of A*] **by** *auto*
obtain *y* **where** *y*: *is-solution y A b* **using** *con* **unfolding** *consistent-def* **by** *blast*
define *f* **where** *f* = (λ*n*::*nat*. *y* + (*of-nat n*) ∗*s x*)
show *?thesis*
proof (*unfold infinite-iff-countable-subset*, *rule exI*[*of - f*], *rule conjI*)
show *inj f* **unfolding** *inj-on-def* **unfolding** *f-def* **using** *x-not-0*
by (*auto simp*: *vec-eq-iff*)
show *range f* ⊆ *solution-set A b*
unfolding *solution-set-rel*[*OF y*]
using *homogeneous-solution-set-subspace* **using** *x* **unfolding** *vec.subspace-def image-def f-def* **by** *fast*
qed
qed

lemma *infinite-solutions-no-homogeneous-imp-dim-solution-set-not-zero-imp*:
fixes *A*::*'a*::{*field*}^*n*::{*mod-type*}^*rows*::{*mod-type*}
assumes *i*: *infinite* (*solution-set A b*)
shows *vec.dim* (*solution-set A 0*) > *0*
using *i independent-and-consistent-imp-card-1 infinite-solution-set-imp-consistent*
by *fastforce*

corollary *infinite-solution-set-no-homogeneous-eq*:
fixes *A*::*'a*::{*field*, *semiring-char-0*}^*n*::{*mod-type*}^*rows*::{*mod-type*}
shows *infinite* (*solution-set A b*) = (*consistent A b* ∧ *vec.dim* (*solution-set A 0*) > *0*)
using *dim-solution-set-not-zero-imp-infinite-solutions-no-homogeneous*
using *infinite-solutions-no-homogeneous-imp-dim-solution-set-not-zero-imp*
using *infinite-solution-set-imp-consistent* **by** *blast*

corollary *infinite-solution-set-no-homogeneous-eq'*:
fixes *A*::*'a*::{*field*, *semiring-char-0*}^*n*::{*mod-type*}^*rows*::{*mod-type*}
shows (∃$_\infty$*x*. *is-solution x A b*) = (*consistent A b* ∧ *vec.dim* (*solution-set A 0*) >

0)
unfolding *infinite-solution-set-no-homogeneous-eq*[*symmetric*] *INFM-iff-infinite* **unfolding** *solution-set-def* **..**

definition *independent-and-consistent A b* = (*consistent A b* ∧ *vec.dim* (*solution-set A 0*) = *0*)
definition *dependent-and-consistent A b* = (*consistent A b* ∧ *vec.dim* (*solution-set A 0*) > *0*)

12.5 Solving systems of linear equations

The following function will solve any system of linear equations. Given a matrix A and a vector b, Firstly it makes use of the funcion *solve-system* to transform the original matrix A and the vector b into another ones in reduced row echelon form. Then, that system will have the same solution than the original one but it is easier to be solved. So we make use of the function *solve-consistent-rref* to obtain one solution of the system.

We will prove that any solution of the system can be rewritten as a linear combination of elements of a basis of the null space plus a particular solution of the system. So the function *solve* will return an option type, depending on the consistency of the system:

- If the system is consistent (so there exists at least one solution), the function will return the *Some* of a pair. In the first component of that pair will be one solution of the system and the second one will be a basis of the null space of the matrix. Hence:
  1. If the system is consistent and independent (so there exists one and only one solution), the pair will consist of the solution and the empty set (this empty set is the basis of the null space).
  2. If the system is consistent and dependent (so there exists more than one solution, maybe an infinite number), the pair will consist of one particular solution and a basis of the null space (which will not be the empty set).
- If the system is inconsistent (so there exists no solution), the function will return *None*.

definition *solve A b* = (*if consistent A b then*
Some (*solve-consistent-rref* (*fst* (*solve-system A b*)) (*snd* (*solve-system A b*)), *basis-null-space A*)
else None)

lemma *solve-code*[*code*]:
shows *solve A b* = (*let GJ-P*=*Gauss-Jordan-PA A*;
P-times-b=*fst*(*GJ-P*) ∗*v b*;

rank-A = (*if A* = *0 then 0 else to-nat* (*GREATEST a. row a* (*snd GJ-P*) ≠ *0*) + *1*);
consistent-Ab = (*rank-A* ≥ (*if* (∃ *a*. (*P-times-b*) $ *a* ≠ *0*) *then* (*to-nat* (*GREATEST a*. (*P-times-b*) $ *a* ≠ *0*) + *1*) *else 0*));
GJ-transpose = *Gauss-Jordan-PA* (*transpose A*);
basis = {*row i* (*fst GJ-transpose*) | *i*. *to-nat i* ≥ *rank-A*}
in (*if consistent-Ab then Some* (*solve-consistent-rref* (*snd GJ-P*) *P-times-b*,*basis*) *else None*))
unfolding *Let-def solve-def*
unfolding *consistent-eq-rank-ge-code*[*unfolded Let-def*,*symmetric*]
unfolding *basis-null-space-def Let-def*
unfolding *P-Gauss-Jordan-def*
unfolding *rank-Gauss-Jordan-code Let-def Gauss-Jordan-PA-eq*
unfolding *solve-system-def Let-def fst-conv snd-conv*
unfolding *Gauss-Jordan-PA-eq* **..**

lemma *consistent-imp-is-solution-solve*:
fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *con*: *consistent A b*
shows *is-solution* (*fst* (*the* (*solve A b*))) *A b*
unfolding *solve-def* **unfolding** *if-P*[*OF con*] *fst-conv* **using** *consistent-imp-is-solution'*[*OF con*]
by *simp*

corollary *consistent-eq-solution-solve*:
fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
shows *consistent A b* = *is-solution* (*fst* (*the* (*solve A b*))) *A b*
by (*metis consistent-def consistent-imp-is-solution-solve*)

lemma *inconsistent-imp-solve-eq-none*:
fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *con*: *inconsistent A b*
shows *solve A b* = *None* **unfolding** *solve-def* **unfolding** *if-not-P*[*OF con*[*unfolded inconsistent-def*]] **..**

corollary *inconsistent-eq-solve-eq-none*:
fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
shows *inconsistent A b* = (*solve A b* = *None*)
unfolding *solve-def* **unfolding** *inconsistent-def* **by** *force*

We demonstrate that all solutions of a system of linear equations can be expressed as a linear combination of the basis of the null space plus a particular solution obtained. The basis and the particular solution are obtained by means of the function *solve A b*

lemma *solution-set-rel-solve*:
fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *con*: *consistent A b*
shows *solution-set A b* = {*fst* (*the* (*solve A b*))} + *vec.span* (*snd* (*the* (*solve A b*)))

proof −
have *s*: *is-solution* (*fst* (*the* (*solve A b*))) *A b* **using** *consistent-imp-is-solution-solve*[*OF con*] **by** *simp*
have *solution-set A b* = {*fst* (*the* (*solve A b*))} + *solution-set A 0* **using** *solution-set-rel*[*OF s*] **.**
also have ... = {*fst* (*the* (*solve A b*))} + *vec.span* (*snd* (*the* (*solve A b*)))
unfolding *set-plus-def solve-def* **unfolding** *if-P*[*OF con*] *snd-conv fst-conv*
proof (*safe*, *simp-all*)
fix *b* **assume** *b* ∈ *solution-set A 0*
thus *b* ∈ *vec.span* (*basis-null-space A*) **unfolding** *null-space-eq-solution-set*[*symmetric*] **using** *basis-null-space*[*of A*] **by** *fast*
next
fix *b*
assume *b*: *b* ∈ *vec.span* (*basis-null-space A*)
thus *b* ∈ *solution-set A 0* **unfolding** *null-space-eq-solution-set*[*symmetric*] **using** *basis-null-space* **by** *blast*
qed
finally show *solution-set A b* = {*fst* (*the* (*solve A b*))} + *vec.span* (*snd* (*the* (*solve A b*))) **.**
qed

lemma *is-solution-eq-in-span-solve*:
fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *con*: *consistent A b*
shows (*is-solution x A b*) = (*x* ∈ {*fst* (*the* (*solve A b*))} + *vec.span* (*snd* (*the* (*solve A b*))))
using *solution-set-rel-solve*[*OF con*] **unfolding** *solution-set-def* **by** *auto*

end

13 Code Generation for Z2

theory *Code-Z2*
imports *HOL−Library.Z2*
begin

Implementation for the field of integer numbers module 2. Experimentally we have checked that the implementation by means of booleans is the fastest one.

code-datatype *0*::*bit* (*1*::*bit*)
code-printing
type-constructor *bit* ⇀ (*SML*) *Bool.bool*
| **constant** *0*::*bit* ⇀ (*SML*) *false*
| **constant** *1*::*bit* ⇀ (*SML*) *true*

code-printing
type-constructor *bit* ⇀ (*Haskell*) *Bool*
| **constant** *0*::*bit* ⇀ (*Haskell*) *False*

| **constant** *1*::*bit* ⇀ (*Haskell*) *True*
| **class-instance** *bit* :: *HOL.equal* => (*Haskell*) –

end

14 Examples of computations over abstract matrices

theory *Examples-Gauss-Jordan-Abstract*
imports
Determinants2
Inverse
System-Of-Equations
Code-Z2
HOL−Library.Code-Target-Numeral
begin

14.1 Transforming a list of lists to an abstract matrix

Definitions to transform a matrix to a list of list and vice versa

definition *vec-to-list* :: *'a^'n*::{*finite*, *enum*} => *'a list*
where *vec-to-list A* = *map* ((*$*) *A*) (*enum-class.enum*::*'n list*)

definition *matrix-to-list-of-list* :: *'a^'n*::{*finite*, *enum*}*^'m*::{*finite*, *enum*} => *'a list list*
where *matrix-to-list-of-list A* = *map* (*vec-to-list*) (*map* ((*$*) *A*) (*enum-class.enum*::*'m list*))

This definition should be equivalent to *vector-def* (in suitable types)

definition *list-to-vec* :: *'a list* => *'a^'n*::{*finite*, *enum*, *mod-type*}
where *list-to-vec xs* = *vec-lambda* (% *i*. *xs* ! (*to-nat i*))

lemma [*code abstract*]: *vec-nth* (*list-to-vec xs*) = (%*i*. *xs* ! (*to-nat i*))
unfolding *list-to-vec-def* **by** *fastforce*

definition *list-of-list-to-matrix* :: *'a list list* => *'a^'n*::{*finite*, *enum*, *mod-type*}*^'m*::{*finite*, *enum*, *mod-type*}
where *list-of-list-to-matrix xs* = *vec-lambda* (%*i*. *list-to-vec* (*xs* ! (*to-nat i*)))

lemma [*code abstract*]: *vec-nth* (*list-of-list-to-matrix xs*) = (%*i*. *list-to-vec* (*xs* ! (*to-nat i*)))
unfolding *list-of-list-to-matrix-def* **by** *auto*

14.2 Examples

The following three lemmas are presented in both this file and in the *Examples-Gauss-Jordan-IArrays* one. They allow a more convenient printing of rational and real numbers after evaluation. They have already been added to the repository version of Isabelle, so after Isabelle2014 they should be removed from here.

lemma [*code-post*]:
int-of-integer (− *1*) = − *1*
by *simp*

lemma [*code-abbrev*]:
(*of-rat* (− *1*) :: *real*) = − *1*
by *simp*

lemma [*code-post*]:
(*of-rat* (− (*1* / *numeral k*)) :: *real*) = − *1* / *numeral k*
(*of-rat* (− (*numeral k* / *numeral l*)) :: *real*) = − *numeral k* / *numeral l*
by (*simp-all add*: *of-rat-divide of-rat-minus*)

14.2.1 Ranks and dimensions

Examples on computing ranks, dimensions of row space, null space and col space and the Gauss Jordan algorithm

value *matrix-to-list-of-list* (*Gauss-Jordan* (*list-of-list-to-matrix* ([[*1*,*0*,*0*,*0*,*0*,*0*],[*0*,*1*,*0*,*0*,*0*,*0*],[*0*,*0*,*0*,*0*,*0*,*0*],[*0*,
value *matrix-to-list-of-list* (*Gauss-Jordan* (*list-of-list-to-matrix* ([[*1*,−*2*,*1*,−*3*,*0*],[*3*,−*6*,*2*,−*7*,*0*]])::*rat*^*5*^*2*))
value *matrix-to-list-of-list* (*Gauss-Jordan* (*list-of-list-to-matrix* ([[*1*,*0*,*0*,*1*,*1*],[*1*,*0*,*1*,*1*,*1*]])::*bit*^*5*^*2*))
value (*reduced-row-echelon-form-upt-k* (*list-of-list-to-matrix* ([[*1*,*0*,*8*],[*0*,*1*,*9*],[*0*,*0*,*0*]])::*real*^*3*^*3*))
3
value *matrix-to-list-of-list* (*Gauss-Jordan* (*list-of-list-to-matrix* [[*Complex 1 1*,*Complex*
1 (− *1*), *Complex 0 0*],[*Complex 2* (− *1*),*Complex 1 3*, *Complex 7 3*]]::*complex*^*3*^*2*))
value *DIM*(*real*^*5*)
value *vec.dimension* (*TYPE*(*bit*)) (*TYPE*(*5*))
value *vec.dimension* (*TYPE*(*real*)) (*TYPE*(*2*))

value *DIM*(*real*^*5*^*4*)
value *row-rank* (*list-of-list-to-matrix* [[*1*,*0*,*0*,*7*,*5*],[*1*,*0*,*4*,*8*,−*1*],[*1*,*0*,*0*,*9*,*8*],[*1*,*2*,*3*,*6*,*5*]]::*real*^*5*^*4*)
value *vec.dim* (*row-space* (*list-of-list-to-matrix* [[*1*,*0*,*0*,*7*,*5*],[*1*,*0*,*4*,*8*,−*1*],[*1*,*0*,*0*,*9*,*8*],[*1*,*2*,*3*,*6*,*5*]]::*real*^*5*^*4*))
value *col-rank* (*list-of-list-to-matrix* [[*1*,*0*,*0*,*7*,*5*],[*1*,*0*,*4*,*8*,−*1*],[*1*,*0*,*0*,*9*,*8*],[*1*,*2*,*3*,*6*,*5*]]::*real*^*5*^*4*)
value *vec.dim* (*col-space* (*list-of-list-to-matrix* [[*1*,*0*,*0*,*7*,*5*],[*1*,*0*,*4*,*8*,−*1*],[*1*,*0*,*0*,*9*,*8*],[*1*,*2*,*3*,*6*,*5*]]::*real*^*5*^*4*))
value *rank* (*list-of-list-to-matrix* [[*1*,*0*,*0*,*7*,*5*],[*1*,*0*,*4*,*8*,−*1*],[*1*,*0*,*0*,*9*,*8*],[*1*,*2*,*3*,*6*,*5*]]::*real*^*5*^*4*)
value *vec.dim* (*null-space* (*list-of-list-to-matrix* [[*1*,*0*,*0*,*7*,*5*],[*1*,*0*,*4*,*8*,−*1*],[*1*,*0*,*0*,*9*,*8*],[*1*,*2*,*3*,*6*,*5*]]::*real*^*5*^*4*))

value *rank* (*list-of-list-to-matrix* [[*Complex 1 1*,*Complex 1* (− *1*), *Complex 0 0*],[*Complex 2* (− *1*),*Complex 1 3*, *Complex 7 3*]]::*complex^3^2*)

14.2.2 Inverse of a matrix

Examples on computing the inverse of matrices

value *let A*=(*list-of-list-to-matrix* [[*1*,*1*,*2*,*4*,*5*,*9*,*8*],[*3*,*0*,*8*,*4*,*5*,*0*,*8*],[*3*,*2*,*0*,*4*,*5*,*9*,*8*],
[*3*,*2*,*8*,*0*,*5*,*9*,*8*] ,[*3*,*2*,*8*,*4*,*0*,*9*,*8*] ,[*3*,*2*,*8*,*4*,*5*,*0*,*8*], [*3*,*2*,*8*,*4*,*5*,*9*,*0*]]::*real^7^7*)
in matrix-to-list-of-list (*P-Gauss-Jordan A*)
value *let A*=(*list-of-list-to-matrix* [[*1*,*1*,*2*,*4*,*5*,*9*,*8*],[*3*,*0*,*8*,*4*,*5*,*0*,*8*],[*3*,*2*,*0*,*4*,*5*,*9*,*8*],
[*3*,*2*,*8*,*0*,*5*,*9*,*8*] ,[*3*,*2*,*8*,*4*,*0*,*9*,*8*] ,[*3*,*2*,*8*,*4*,*5*,*0*,*8*], [*3*,*2*,*8*,*4*,*5*,*9*,*0*]]::*real^7^7*) *in*
matrix-to-list-of-list (*A* ∗∗ (*P-Gauss-Jordan A*))
value *let A*=(*list-of-list-to-matrix* [[*1*,*1*,*2*,*4*,*5*,*9*,*8*],[*3*,*0*,*8*,*4*,*5*,*0*,*8*],[*3*,*2*,*0*,*4*,*5*,*9*,*8*],
[*3*,*2*,*8*,*0*,*5*,*9*,*8*] ,[*3*,*2*,*8*,*4*,*0*,*9*,*8*] ,[*3*,*2*,*8*,*4*,*5*,*0*,*8*], [*3*,*2*,*8*,*4*,*5*,*9*,*0*]]::*real^7^7*)
in (*inverse-matrix A*)
value *let A*=(*list-of-list-to-matrix* [[*1*,*1*,*2*,*4*,*5*,*9*,*8*],[*3*,*0*,*8*,*4*,*5*,*0*,*8*],[*3*,*2*,*0*,*4*,*5*,*9*,*8*],
[*3*,*2*,*8*,*0*,*5*,*9*,*8*] ,[*3*,*2*,*8*,*4*,*0*,*9*,*8*] ,[*3*,*2*,*8*,*4*,*5*,*0*,*8*], [*3*,*2*,*8*,*4*,*5*,*9*,*0*]]::*real^7^7*)
in matrix-to-list-of-list (*the* (*inverse-matrix A*))
value *let A*=(*list-of-list-to-matrix* [[*1*,*1*,*1*,*1*,*1*,*1*,*1*],[*2*,*2*,*2*,*2*,*2*,*2*,*2*],[*3*,*2*,*0*,*4*,*5*,*9*,*8*],
[*3*,*2*,*8*,*0*,*5*,*9*,*8*] ,[*3*,*2*,*8*,*4*,*0*,*9*,*8*] ,[*3*,*2*,*8*,*4*,*5*,*0*,*8*], [*3*,*2*,*8*,*4*,*5*,*9*,*0*]]::*real^7^7*)
in (*inverse-matrix A*)
value *let A*=(*list-of-list-to-matrix* [[*Complex 1 1*,*Complex 1* (− *1*), *Complex 0 0*],[*Complex 1 1*,*Complex 1* (− *1*), *Complex 8 0*],[*Complex 2* (− *1*),*Complex 1 3*, *Complex 7 3*]]::*complex^3^3*)
in matrix-to-list-of-list (*the* (*inverse-matrix A*))

14.2.3 Determinant of a matrix

Examples on computing determinants of matrices

value (*let A = list-of-list-to-matrix*[[*1*,*2*,*7*,*8*,*9*],[*3*,*4*,*12*,*10*,*7*],[−*5*,*4*,*8*,*7*,*4*],[*0*,*1*,*2*,*4*,*8*],[*9*,*8*,*7*,*13*,*11*]]::*real^5^
in det A)
value *det* (*list-of-list-to-matrix* ([[*1*,*0*,*0*],[*0*,*1*,*0*],[*0*,*0*,*1*]])::*real^3^3*)
value *det* (*list-of-list-to-matrix* ([[*1*,*8*,*9*,*1*,*47*],[*7*,*2*,*2*,*5*,*9*],[*3*,*2*,*7*,*7*,*4*],[*9*,*8*,*7*,*5*,*1*],[*1*,*2*,*6*,*4*,*5*]])::*rat^5^5*)

14.2.4 Bases of the fundamental subspaces

Examples on computing basis for null space, row space, column space and left null space

value *let A* = (*list-of-list-to-matrix* ([[*1*,*3*,−*2*,*0*,*2*,*0*],[*2*,*6*,−*5*,−*2*,*4*,−*3*],[*0*,*0*,*5*,*10*,*0*,*15*],[*2*,*6*,*0*,*8*,*4*,*18*]])::*real*

in vec-to-list‘ (*basis-null-space A*)
value *let A* = (*list-of-list-to-matrix* ([[*3*,*4*,*0*,*7*],[*1*,−*5*,*2*,−*2*],[−*1*,*4*,*0*,*3*],[*1*,−*1*,*2*,*2*]])::*real^4^4*)

in vec-to-list‘ (*basis-null-space A*)

value *let A* = (*list-of-list-to-matrix* ([[*1*,*3*,−*2*,*0*,*2*,*0*],[*2*,*6*,−*5*,−*2*,*4*,−*3*],[*0*,*0*,*5*,*10*,*0*,*15*],[*2*,*6*,*0*,*8*,*4*,*18*]])::*real*

in vec-to-list‘ (basis-row-space A)
value *let A = (list-of-list-to-matrix ([[3,4,0,7],[1,−5,2,−2],[−1,4,0,3],[1,−1,2,2]])::real^4^4)*

in vec-to-list‘ (basis-row-space A)

value *let A = (list-of-list-to-matrix ([[1,3,−2,0,2,0],[2,6,−5,−2,4,−3],[0,0,5,10,0,15],[2,6,0,8,4,18]])::real*

in vec-to-list‘ (basis-col-space A)
value *let A = (list-of-list-to-matrix ([[3,4,0,7],[1,−5,2,−2],[−1,4,0,3],[1,−1,2,2]])::real^4^4)*

in vec-to-list‘ (basis-col-space A)

value *let A = (list-of-list-to-matrix ([[1,3,−2,0,2,0],[2,6,−5,−2,4,−3],[0,0,5,10,0,15],[2,6,0,8,4,18]])::real*

in vec-to-list‘ (basis-left-null-space A)
value *let A = (list-of-list-to-matrix ([[3,4,0,7],[1,−5,2,−2],[−1,4,0,3],[1,−1,2,2]])::real^4^4)*

in vec-to-list‘ (basis-left-null-space A)

14.2.5 Consistency and inconsistency

Examples on checking the consistency/inconsistency of a system of equations

value *independent-and-consistent (list-of-list-to-matrix ([[1,0,0],[0,1,0],[0,0,1],[0,0,0],[0,0,0]])::real^3^5)*
(list-to-vec([2,3,4,0,0])::real^5)
value *consistent (list-of-list-to-matrix ([[1,0,0],[0,1,0],[0,0,1],[0,0,0],[0,0,0]])::real^3^5)*
(list-to-vec([2,3,4,0,0])::real^5)
value *inconsistent (list-of-list-to-matrix ([[1,0,0],[0,1,0],[3,0,1],[0,7,0],[0,0,9]])::real^3^5)*
(list-to-vec([2,0,4,0,0])::real^5)
value *dependent-and-consistent (list-of-list-to-matrix ([[1,0,0],[0,1,0]])::real^3^2)*
(list-to-vec([3,4])::real^2)
value *independent-and-consistent (mat 1::real^3^3) (list-to-vec([3,4,5])::real^3)*

14.2.6 Solving systems of linear equations

Examples on solving linear systems.

definition *print-result-solve*
where *print-result-solve A = (if A = None then None else Some (vec-to-list (fst (the A)), vec-to-list‘ (snd (the A))))*

value *let A = (list-of-list-to-matrix [[4,5,8],[9,8,7],[4,6,1]]::real^3^3);*
b=(list-to-vec [4,5,8]::real^3)
in (print-result-solve (solve A b))

value *let A = (list-of-list-to-matrix [[0,0,0],[0,0,0],[0,0,1]]::real^3^3);*
b=(list-to-vec [4,5,0]::real^3)
in (print-result-solve (solve A b))

value *let A = (list-of-list-to-matrix [[3,2,5,2,7],[6,4,7,4,5],[3,2,−1,2,−11],[6,4,1,4,−13]]::real^5^4);*

b=(list-to-vec [0,0,0,0]::real^4)
in (print-result-solve (solve A b))

value *let A = (list-of-list-to-matrix [[1,2,1],[−2,−3,−1],[2,4,2]]::real^3^3);*
b=(list-to-vec [−2,1,−4]::real^3)
in (print-result-solve (solve A b))

value *let A = (list-of-list-to-matrix [[1,1,−4,10],[3,−2,−2,6]]::real^4^2);*
b=(list-to-vec [24,15]::real^2)
in (print-result-solve (solve A b))

end

15 IArrays Addenda

theory *IArray-Addenda*
imports
HOL−Library.IArray
begin

15.1 Some previous instances

instantiation *iarray* :: (*plus*) *plus*
begin
definition *plus-iarray* :: *'a iarray ⇒ 'a iarray ⇒ 'a iarray*
where *plus-iarray A B = IArray.of-fun (λn. A!!n + B !! n) (IArray.length A)*
instance proof qed
end

instantiation *iarray* :: (*minus*) *minus*
begin
definition *minus-iarray* :: *'a iarray ⇒ 'a iarray ⇒ 'a iarray*
where *minus-iarray A B = IArray.of-fun (λn. A!!n − B !! n) (IArray.length A)*
instance proof qed
end

15.2 Some previous definitions and properties for IArrays

15.3 Code generation

end

16 Matrices as nested IArrays

theory *Matrix-To-IArray*
imports
Rank-Nullity-Theorem.Mod-Type

Elementary-Operations
IArray-Addenda
begin

16.1 Isomorphism between matrices implemented by vecs and matrices implemented by iarrays

16.1.1 Isomorphism between vec and iarray

definition *vec-to-iarray* :: *'a^'n::{mod-type}* ⇒ *'a iarray*
where *vec-to-iarray A* = *IArray.of-fun* (λ*i*. *A* $ (*from-nat i*)) (*CARD*(*'n*))

definition *iarray-to-vec* :: *'a iarray* ⇒ *'a^'n::{mod-type}*
where *iarray-to-vec A* = (χ *i*. *A* !! (*to-nat i*))

lemma *vec-to-iarray-nth*:
fixes *A::'a^'n::{finite, mod-type}*
assumes *i*: *i*<*CARD*(*'n*)
shows (*vec-to-iarray A*) !! *i* = *A* $ (*from-nat i*)
unfolding *vec-to-iarray-def* **using** *of-fun-nth*[*OF i*] **.**

lemma *vec-to-iarray-nth'*:
fixes *A::'a^'n::{mod-type}*
shows (*vec-to-iarray A*) !! (*to-nat i*) = *A* $ *i*
proof −
have *to-nat-less-card*: *to-nat i*<*CARD*(*'n*) **using** *bij-to-nat*[**where** *?'a*=*'n*] **unfolding** *bij-betw-def* **by** *fastforce*
show *?thesis* **unfolding** *vec-to-iarray-def* **unfolding** *of-fun-nth*[*OF to-nat-less-card*] *from-nat-to-nat-id* **..**
qed

lemma *iarray-to-vec-nth*:
shows (*iarray-to-vec A*) $ *i* = *A* !! (*to-nat i*)
unfolding *iarray-to-vec-def* **by** *simp*

lemma *vec-to-iarray-morph*:
fixes *A::'a^'n::{mod-type}*
shows (*A* = *B*) = (*vec-to-iarray A* = *vec-to-iarray B*)
by (*metis vec-eq-iff vec-to-iarray-nth'*)

lemma *inj-vec-to-iarray*:
shows *inj vec-to-iarray*
using *vec-to-iarray-morph* **unfolding** *inj-on-def* **by** *blast*

lemma *iarray-to-vec-vec-to-iarray*:
fixes *A::'a^'n::{mod-type}*
shows *iarray-to-vec* (*vec-to-iarray A*)=*A*

proof (*unfold vec-to-iarray-def iarray-to-vec-def*, *vector*, *auto*)
fix *i*::*'n*
have *to-nat i*<*CARD*(*'n*) **using** *bij-to-nat*[**where** *?'a*=*'n*] **unfolding** *bij-betw-def*
by *auto*
thus *map* (*λi. A $ from-nat i*) [*0*..<*CARD*(*'n*)] ! *to-nat i* = *A $ i* **by** *simp*
qed

lemma *vec-to-iarray-iarray-to-vec*:
assumes *length-eq*: *IArray.length A* = *CARD*(*'n*::{*mod-type*})
shows *vec-to-iarray* (*iarray-to-vec A*::*'a*^*'n*::{*mod-type*}) = *A*
proof (*unfold vec-to-iarray-def iarray-to-vec-def*, *vector*, *auto*)
obtain *xs* **where** *xs*: *A* = *IArray xs* **by** (*metis iarray.exhaust*)
show *IArray* (*map* (*λi. IArray.list-of A* ! *to-nat* (*from-nat i*::*'n*)) [*0*..<*CARD*(*'n*)])
= *A*
proof(*unfold xs iarray.inject list-eq-iff-nth-eq*, *auto*)
show *CARD*(*'n*) = *length xs* **using** *length-eq* **unfolding** *xs* **by** *simp*
fix *i* **assume** *i*: *i* < *CARD*(*'n*)
show *xs* ! *to-nat* (*from-nat i*::*'n*) = *xs* ! *i* **unfolding** *to-nat-from-nat-id*[*OF i*]
..
qed
qed

lemma *length-vec-to-iarray*:
fixes *xa*::*'a*^*'n*::{*mod-type*}
shows *IArray.length* (*vec-to-iarray xa*) = *CARD*(*'n*)
unfolding *vec-to-iarray-def* **by** *simp*

16.1.2 Isomorphism between matrix and nested iarrays

definition *matrix-to-iarray* :: *'a*^*'n*::{*mod-type*}^*'m*::{*mod-type*} => *'a iarray iarray*
where *matrix-to-iarray A* = *IArray* (*map* (*vec-to-iarray* ∘ (($) *A*) ∘ (*from-nat*::*nat*=>*'m*))
[*0*..<*CARD*(*'m*)])

definition *iarray-to-matrix* :: *'a iarray iarray* ⇒ *'a*^*'n*::{*mod-type*}^*'m*::{*mod-type*}
where *iarray-to-matrix A* = (*χ i j. A* !! (*to-nat i*) !! (*to-nat j*))

lemma *matrix-to-iarray-morph*:
fixes *A*::*'a*^*'n*::{*mod-type*}^*'m*::{*mod-type*}
shows (*A* = *B*) = (*matrix-to-iarray A* = *matrix-to-iarray B*)
unfolding *matrix-to-iarray-def* **apply** *simp*
unfolding *forall-from-nat-rw*[*of λx. vec-to-iarray* (*A $ x*) = *vec-to-iarray* (*B $*
x)]
by (*metis from-nat-to-nat-id vec-eq-iff vec-to-iarray-morph*)

lemma *matrix-to-iarray-eq-of-fun*:
fixes *A*::*'a*^*columns*::{*mod-type*}^*rows*::{*mod-type*}
assumes *vec-eq-f*: ∀ *i. vec-to-iarray* (*A $ i*) = *f* (*to-nat i*)
and *n-eq-length*: *n*=*IArray.length* (*matrix-to-iarray A*)

shows *matrix-to-iarray A = IArray.of-fun f n*
proof (*unfold IArray.of-fun-def matrix-to-iarray-def iarray.inject list-eq-iff-nth-eq*, *auto*)
show ∗: *CARD('rows) = n* **using** *n-eq-length* **unfolding** *matrix-to-iarray-def* **by** *auto*
fix *i* **assume** *i*: *i < CARD('rows)*
hence *i-less-n*: *i<n* **using** ∗ *i* **by** *simp*
show *vec-to-iarray (A $ from-nat i) = map f [0..<n] ! i*
using *vec-eq-f* **using** *i-less-n*
by (*simp*, *unfold to-nat-from-nat-id*[*OF i*], *simp*)
qed

lemma *map-vec-to-iarray-rw*[*simp*]:
fixes *A*::*'a*^*'columns*::{*mod-type*}^*'rows*::{*mod-type*}
shows *map* (λ*x*. *vec-to-iarray* (*A $ from-nat x*)) [*0..<CARD('rows)*] ! *to-nat i = vec-to-iarray* (*A $ i*)
proof −
have *i-less-card*: *to-nat i < CARD('rows)* **using** *bij-to-nat*[**where** *?'a='rows*] **unfolding** *bij-betw-def* **by** *fastforce*
hence *map* (λ*x*. *vec-to-iarray* (*A $ from-nat x*)) [*0..<CARD('rows)*] ! *to-nat i = vec-to-iarray* (*A $ from-nat* (*to-nat i*)) **by** *simp*
also have ... = *vec-to-iarray* (*A $ i*) **unfolding** *from-nat-to-nat-id* **..**
finally show *?thesis* **.**
qed

lemma *matrix-to-iarray-nth*:
matrix-to-iarray A !! *to-nat i* !! *to-nat j = A $ i $ j*
unfolding *matrix-to-iarray-def o-def* **using** *vec-to-iarray-nth'* **by** *auto*

lemma *vec-matrix*: *vec-to-iarray* (*A$i*) = (*matrix-to-iarray A*) !! (*to-nat i*)
unfolding *matrix-to-iarray-def o-def* **by** *fastforce*

lemma *iarray-to-matrix-matrix-to-iarray*:
fixes *A*::*'a*^*'columns*::{*mod-type*}^*'rows*::{*mod-type*}
shows *iarray-to-matrix* (*matrix-to-iarray A*)=*A* **unfolding** *matrix-to-iarray-def iarray-to-matrix-def o-def*
by (*vector*, *auto*, *metis IArray.sub-def vec-to-iarray-nth'*)

16.2 Definition of operations over matrices implemented by iarrays

definition *mult-iarray* :: *'a*::{*times*} *iarray* => *'a* => *'a iarray*
where *mult-iarray A q = IArray.of-fun* (λ*n*. *q* ∗ *A*!!*n*) (*IArray.length A*)

definition *row-iarray* :: *nat* => *'a iarray iarray* => *'a iarray*
where *row-iarray k A = A* !! *k*

definition *column-iarray* :: *nat* => *'a iarray iarray* => *'a iarray*

where *column-iarray k A = IArray.of-fun* (λ*m*. *A* !! *m* !! *k*) (*IArray.length A*)

definition *nrows-iarray* :: *'a iarray iarray => nat*
where *nrows-iarray A = IArray.length A*

definition *ncols-iarray* :: *'a iarray iarray => nat*
where *ncols-iarray A = IArray.length* (*A*!!*0*)

definition *rows-iarray A* = {*row-iarray i A* | *i*. *i* ∈ {..<*nrows-iarray A*}}
definition *columns-iarray A* = {*column-iarray i A* | *i*. *i* ∈ {..<*ncols-iarray A*}}

definition *tabulate2* :: *nat => nat => (nat => nat => 'a) => 'a iarray iarray*
where *tabulate2 m n f = IArray.of-fun* (λ*i*. *IArray.of-fun* (*f i*) *n*) *m*

definition *transpose-iarray* :: *'a iarray iarray => 'a iarray iarray*
where *transpose-iarray A = tabulate2* (*ncols-iarray A*) (*nrows-iarray A*) (λ*a b*. *A*!!*b*!!*a*)

definition *matrix-matrix-mult-iarray* :: *'a*::{*times*, *comm-monoid-add*} *iarray iarray => 'a iarray iarray => 'a iarray iarray* (**infixl** ‹∗∗*i*› *70*)
where *A* ∗∗*i B = tabulate2* (*nrows-iarray A*) (*ncols-iarray B*) (λ*i j*. *sum* (λ*k*. ((*A*!!*i*)!!*k*) ∗ ((*B*!!*k*)!!*j*)) {*0*..<*ncols-iarray A*})

definition *matrix-vector-mult-iarray* :: *'a*::{*semiring-1*} *iarray iarray => 'a iarray => 'a iarray* (**infixl** ‹∗*iv*› *70*)
where *A* ∗*iv x = IArray.of-fun* (λ*i*. *sum* (λ*j*. ((*A*!!*i*)!!*j*) ∗ (*x*!!*j*)) {*0*..<*IArray.length x*}) (*nrows-iarray A*)

definition *vector-matrix-mult-iarray* :: *'a*::{*semiring-1*} *iarray => 'a iarray iarray => 'a iarray* (**infixl** ‹*v*∗*i*› *70*)
where *x v*∗*i A = IArray.of-fun* (λ*j*. *sum* (λ*i*. (*x*!!*i*) ∗ ((*A*!!*i*)!!*j*)) {*0*..<*IArray.length x*}) (*ncols-iarray A*)

definition *mat-iarray* :: *'a*::{*zero*} *=> nat => 'a iarray iarray*
where *mat-iarray k n = tabulate2 n n* (λ *i j*. *if i = j then k else 0*)

definition *is-zero-iarray* :: *'a*::{*zero*} *iarray* ⇒ *bool*
where *is-zero-iarray A = IArray.all* (λ*i*. *A* !! *i = 0*) (*IArray*[*0*..<*IArray.length A*])

16.2.1 Properties of previous definitions

lemma *is-zero-iarray-eq-iff*:
fixes *A*::*'a*::{*zero*}^*'n*::{*mod-type*}
shows (*A = 0*) = (*is-zero-iarray* (*vec-to-iarray A*))
proof (*auto*)
show *is-zero-iarray* (*vec-to-iarray 0*) **by** (*simp add*: *vec-to-iarray-def is-zero-iarray-def Option.is-none-def find-None-iff*)
show *is-zero-iarray* (*vec-to-iarray A*) ⟹ *A = 0*

proof (*simp add*: *vec-to-iarray-def is-zero-iarray-def Option.is-none-def find-None-iff vec-eq-iff*, *clarify*)
fix *i*::*'n*
assume $\forall i{\in}\{0..{<}CARD('n)\}$. *A* $ *mod-type-class.from-nat i* = *0*
hence *eq-zero*: $\forall x{<}CARD('n)$. *A* $ *from-nat x* = *0* **by** *force*
have *to-nat i*$<$*CARD*(*'n*) **using** *bij-to-nat*[**where** *?'a*=*'n*] **unfolding** *bij-betw-def* **by** *fastforce*
hence *A* $ (*from-nat* (*to-nat i*)) = *0* **using** *eq-zero* **by** *blast*
thus *A* $ *i* = *0* **unfolding** *from-nat-to-nat-id* **.**
qed
qed

lemma *mult-iarray-works*:
assumes *a*$<$*IArray.length A* **shows** *mult-iarray A q* !! *a* = *q* ∗ *A*!!*a*
unfolding *mult-iarray-def*
unfolding *IArray.of-fun-def* **unfolding** *sub-def*
using *assms* **by** *simp*

lemma *length-eq-card-rows*:
fixes *A*::*'a*^{}*columns*::{*mod-type*}^{}*rows*::{*mod-type*}
shows *IArray.length* (*matrix-to-iarray A*) = *CARD*(*'rows*)
unfolding *matrix-to-iarray-def* **by** *auto*

lemma *nrows-eq-card-rows*:
fixes *A*::*'a*^{}*columns*::{*mod-type*}^{}*rows*::{*mod-type*}
shows *nrows-iarray* (*matrix-to-iarray A*) = *CARD*(*'rows*)
unfolding *nrows-iarray-def length-eq-card-rows* **..**

lemma *length-eq-card-columns*:
fixes *A*::*'a*^{}*columns*::{*mod-type*}^{}*rows*::{*mod-type*}
shows *IArray.length* (*matrix-to-iarray A* !! *0*) = *CARD* (*'columns*)
unfolding *matrix-to-iarray-def o-def vec-to-iarray-def* **by** *simp*

lemma *ncols-eq-card-columns*:
fixes *A*::*'a*^{}*columns*::{*mod-type*}^{}*rows*::{*mod-type*}
shows *ncols-iarray* (*matrix-to-iarray A*) = *CARD*(*'columns*)
unfolding *ncols-iarray-def length-eq-card-columns* **..**

lemma *matrix-to-iarray-nrows*:
fixes *A*::*'a*^{}*columns*::{*mod-type*}^{}*rows*::{*mod-type*}
shows *nrows A* = *nrows-iarray* (*matrix-to-iarray A*)
unfolding *nrows-def nrows-eq-card-rows* **..**

lemma *matrix-to-iarray-ncols*:
fixes *A*::*'a*^{}*columns*::{*mod-type*}^{}*rows*::{*mod-type*}
shows *ncols A* = *ncols-iarray* (*matrix-to-iarray A*)
unfolding *ncols-def ncols-eq-card-columns* **..**

lemma *vec-to-iarray-row*[*code-unfold*]: *vec-to-iarray* (*row i A*) = *row-iarray* (*to-nat*

i) (*matrix-to-iarray A*)
unfolding *row-def row-iarray-def vec-to-iarray-def*
by (*auto*, *metis IArray.sub-def IArray.of-fun-def vec-matrix vec-to-iarray-def*)

lemma *vec-to-iarray-row'*: *vec-to-iarray* (*row i A*) = (*matrix-to-iarray A*) !! (*to-nat i*)
unfolding *row-def vec-to-iarray-def*
by (*auto*, *metis IArray.sub-def IArray.of-fun-def vec-matrix vec-to-iarray-def*)

lemma *vec-to-iarray-column*[*code-unfold*]: *vec-to-iarray* (*column i A*) = *column-iarray* (*to-nat i*) (*matrix-to-iarray A*)
unfolding *column-def vec-to-iarray-def column-iarray-def length-eq-card-rows*
by (*auto*, *metis IArray.sub-def from-nat-not-eq vec-matrix vec-to-iarray-nth'*)

lemma *vec-to-iarray-column'*:
assumes *k*: *k*<*ncols A*
shows (*vec-to-iarray* (*column* (*from-nat k*) *A*)) = (*column-iarray k* (*matrix-to-iarray A*))
unfolding *vec-to-iarray-column* **unfolding** *to-nat-from-nat-id*[*OF k*[*unfolded ncols-def*]]
..

lemma *column-iarray-nth*:
assumes *i*: *i*<*nrows-iarray A*
shows *column-iarray j A* !! *i* = *A* !! *i* !! *j*
proof −
have *column-iarray j A* !! *i* = *map* (λ*m*. *A* !! *m* !! *j*) [*0*..<*IArray.length A*] ! *i*
unfolding *column-iarray-def* **by** *auto*
also have ... = (λ*m*. *A* !! *m* !! *j*) ([*0*..<*IArray.length A*] ! *i*) **using** *i nth-map* **unfolding** *nrows-iarray-def* **by** *auto*
also have ... = (λ*m*. *A* !! *m* !! *j*) (*i*) **using** *nth-upt*[*of 0 i IArray.length A*] *i* **unfolding** *nrows-iarray-def* **by** *simp*
finally show *?thesis* **.**
qed

lemma *vec-to-iarray-rows*: *vec-to-iarray‘* (*rows A*) = *rows-iarray* (*matrix-to-iarray A*)
unfolding *rows-def* **unfolding** *rows-iarray-def*
apply (*auto simp add*: *vec-to-iarray-row to-nat-less-card nrows-eq-card-rows*)
by (*unfold image-def*, *auto*, *metis from-nat-not-eq vec-to-iarray-row*)

lemma *vec-to-iarray-columns*: *vec-to-iarray‘* (*columns A*) = *columns-iarray* (*matrix-to-iarray A*)
unfolding *columns-def* **unfolding** *columns-iarray-def*
apply(*auto simp add*: *ncols-eq-card-columns to-nat-less-card vec-to-iarray-column*)
by (*unfold image-def*, *auto*, *metis from-nat-not-eq vec-to-iarray-column*)

16.3 Definition of elementary operations

definition *interchange-rows-iarray* :: *'a iarray iarray => nat => nat => 'a iarray iarray*
where *interchange-rows-iarray A a b = IArray.of-fun* (*λn. if n=a then A!!b else if n=b then A!!a else A!!n*) (*IArray.length A*)

definition *mult-row-iarray* :: *'a::{times} iarray iarray => nat => 'a => 'a iarray iarray*
where *mult-row-iarray A a q = IArray.of-fun* (*λn. if n=a then mult-iarray* (*A!!a*) *q else A!!n*) (*IArray.length A*)

definition *row-add-iarray* :: *'a::{plus, times} iarray iarray => nat => nat => 'a => 'a iarray iarray*
where *row-add-iarray A a b q = IArray.of-fun* (*λn. if n=a then A!!a + mult-iarray* (*A!!b*) *q else A!!n*) (*IArray.length A*)

definition *interchange-columns-iarray* :: *'a iarray iarray => nat => nat => 'a iarray iarray*
where *interchange-columns-iarray A a b = tabulate2* (*nrows-iarray A*) (*ncols-iarray A*) (*λi j. if j = a then A !! i !! b else if j = b then A !! i !! a else A !! i !! j*)

definition *mult-column-iarray* :: *'a::{times} iarray iarray => nat => 'a => 'a iarray iarray*
where *mult-column-iarray A n q = tabulate2* (*nrows-iarray A*) (*ncols-iarray A*) (*λi j. if j = n then A !! i !! j * q else A !! i !! j*)

definition *column-add-iarray* :: *'a::{plus, times} iarray iarray => nat => nat => 'a => 'a iarray iarray*
where *column-add-iarray A n m q = tabulate2* (*nrows-iarray A*) (*ncols-iarray A*) (*λi j. if j = n then A !! i !! n + A !! i !! m * q else A !! i !! j*)

16.3.1 Code generator

lemma *vec-to-iarray-plus*[*code-unfold*]: *vec-to-iarray* (*a + b*) = (*vec-to-iarray a*) + (*vec-to-iarray b*)
unfolding *vec-to-iarray-def*
unfolding *plus-iarray-def* **by** *auto*

lemma *matrix-to-iarray-plus*[*code-unfold*]: *matrix-to-iarray* (*A + B*) = (*matrix-to-iarray A*) + (*matrix-to-iarray B*)
unfolding *matrix-to-iarray-def o-def*
by (*simp add*: *plus-iarray-def vec-to-iarray-plus*)

lemma *matrix-to-iarray-mat*[*code-unfold*]:
matrix-to-iarray (*mat k* ::*'a::{zero}*^*'n::{mod-type}*^*'n::{mod-type}*) = *mat-iarray k CARD*(*'n::{mod-type}*)
unfolding *matrix-to-iarray-def o-def vec-to-iarray-def mat-def mat-iarray-def tabulate2-def*
using *from-nat-eq-imp-eq* **by** *fastforce*

lemma *matrix-to-iarray-transpose*[*code-unfold*]:
shows *matrix-to-iarray* (*transpose A*) = *transpose-iarray* (*matrix-to-iarray A*)
unfolding *matrix-to-iarray-def transpose-def transpose-iarray-def*
o-def tabulate2-def nrows-iarray-def ncols-iarray-def vec-to-iarray-def
by *auto*

lemma *matrix-to-iarray-matrix-matrix-mult*[*code-unfold*]:
fixes *A*::*'a*::{*semiring-1*}^*'m*::{*mod-type*}^*'n*::{*mod-type*} **and** *B*::*'a*^*'b*::{*mod-type*}^*'m*::{*mod-type*}
shows *matrix-to-iarray* (*A* ** *B*) = (*matrix-to-iarray A*) **i* (*matrix-to-iarray B*)
unfolding *matrix-to-iarray-def matrix-matrix-mult-iarray-def matrix-matrix-mult-def*

unfolding *o-def tabulate2-def nrows-iarray-def ncols-iarray-def vec-to-iarray-def*
using *sum.reindex*[*of from-nat*::*nat*=>*'m*] **using** *bij-from-nat* **unfolding** *bij-betw-def*
by *fastforce*

lemma *vec-to-iarray-matrix-matrix-mult*[*code-unfold*]:
fixes *A*::*'a*::{*semiring-1*}^*'m*::{*mod-type*}^*'n*::{*mod-type*} **and** *x*::*'a*^*'m*::{*mod-type*}
shows *vec-to-iarray* (*A* *v *x*) = (*matrix-to-iarray A*) **iv* (*vec-to-iarray x*)
unfolding *matrix-vector-mult-iarray-def matrix-vector-mult-def*
unfolding *o-def tabulate2-def nrows-iarray-def ncols-iarray-def matrix-to-iarray-def vec-to-iarray-def*
using *sum.reindex*[*of from-nat*::*nat*=>*'m*] **using** *bij-from-nat* **unfolding** *bij-betw-def*
by *fastforce*

lemma *vec-to-iarray-vector-matrix-mult*[*code-unfold*]:
fixes *A*::*'a*::{*semiring-1*}^*'m*::{*mod-type*}^*'n*::{*mod-type*} **and** *x*::*'a*^*'n*::{*mod-type*}
shows *vec-to-iarray* (*x v** *A*) = (*vec-to-iarray x*) *v*i* (*matrix-to-iarray A*)
unfolding *vector-matrix-mult-def vector-matrix-mult-iarray-def*
unfolding *o-def tabulate2-def nrows-iarray-def ncols-iarray-def matrix-to-iarray-def vec-to-iarray-def*
proof (*auto*)
fix *xa*
have (*UNIV*::*'n set*) = *from-nat* `{*0*..<*CARD*(*'n*)} **using** *bij-from-nat*[**where** *?'a*=*'n*] **unfolding** *bij-betw-def* **by** *fast*
show (∑ *i*∈*UNIV*. *x* $ *i* * *A* $ *i* $ *from-nat xa*) = (∑ *i* = *0*..<*CARD*(*'n*). *x*$ *from-nat i* * *A* $ *from-nat i* $ *from-nat xa*)
using *sum.reindex*[*of from-nat*::*nat*=>*'n*] **using** *bij-from-nat*[**where** *?'a*=*'n*]
unfolding *bij-betw-def* **by** *force*
qed

lemma *matrix-to-iarray-interchange-rows*[*code-unfold*]:
fixes *A*::*'a*::{*semiring-1*}^*columns*::{*mod-type*}^*rows*::{*mod-type*}
shows *matrix-to-iarray* (*interchange-rows A i j*) = *interchange-rows-iarray* (*matrix-to-iarray A*) (*to-nat i*) (*to-nat j*)

proof (*unfold matrix-to-iarray-def interchange-rows-iarray-def o-def map-vec-to-iarray-rw*, *auto*)
fix *x* **assume** *x-less-card*: $x < CARD('rows)$
and *x-not-j*: $x \neq to\text{-}nat\ j$ **and** *x-not-i*: $x \neq to\text{-}nat\ i$
show *vec-to-iarray* (*interchange-rows A i j* $ *from-nat x*) = *vec-to-iarray* (*A* $ *from-nat x*)
by (*metis interchange-rows-preserves to-nat-from-nat-id x-less-card x-not-i x-not-j*)
qed

lemma *matrix-to-iarray-mult-row*[*code-unfold*]:
fixes *A*::'*a*::{*semiring-1*}^'*columns*::{*mod-type*}^'*rows*::{*mod-type*}
shows *matrix-to-iarray* (*mult-row A i q*) = *mult-row-iarray* (*matrix-to-iarray A*) (*to-nat i*) *q*
unfolding *matrix-to-iarray-def mult-row-iarray-def o-def*
unfolding *mult-iarray-def vec-to-iarray-def mult-row-def* **apply** *auto*
proof −
fix *i x*
assume *i-contr*:$i \neq to\text{-}nat\ (from\text{-}nat\ i::'rows)$ **and** $x < CARD('columns)$
and $i<CARD('rows)$
hence $i = to\text{-}nat\ (from\text{-}nat\ i::'rows)$ **using** *to-nat-from-nat-id* **by** *fastforce*
thus $q * A\ \$\ from\text{-}nat\ i\ \$\ from\text{-}nat\ x = A\ \$\ from\text{-}nat\ i\ \$\ from\text{-}nat\ x$
using *i-contr* **by** *contradiction*
qed

lemma *matrix-to-iarray-row-add*[*code-unfold*]:
fixes *A*::'*a*::{*semiring-1*}^'*columns*::{*mod-type*}^'*rows*::{*mod-type*}
shows *matrix-to-iarray* (*row-add A i j q*) = *row-add-iarray* (*matrix-to-iarray A*) (*to-nat i*) (*to-nat j*) *q*
proof (*unfold matrix-to-iarray-def row-add-iarray-def o-def*, *auto*)
show *vec-to-iarray* (*row-add A i j q* $ *i*) = *vec-to-iarray* (*A* $ *i*) + *mult-iarray* (*vec-to-iarray* (*A* $ *j*)) *q*
unfolding *mult-iarray-def vec-to-iarray-def* **unfolding** *plus-iarray-def row-add-def*
by *auto*
fix *ia* **assume** *ia-not-i*: $ia \neq to\text{-}nat\ i$ **and** *ia-card*: $ia < CARD('rows)$
have *from-nat-ia-not-i*: $from\text{-}nat\ ia \neq i$
proof (*rule ccontr*)
assume ¬ $from\text{-}nat\ ia \neq i$ **hence** $from\text{-}nat\ ia = i$ **by** *simp*
hence $to\text{-}nat\ (from\text{-}nat\ ia::'rows) = to\text{-}nat\ i$ **by** *simp*
hence $ia=to\text{-}nat\ i$ **using** *to-nat-from-nat-id ia-card* **by** *fastforce*
thus *False* **using** *ia-not-i* **by** *contradiction*
qed
show *vec-to-iarray* (*row-add A i j q* $ *from-nat ia*) = *vec-to-iarray* (*A* $ *from-nat ia*)
using *ia-not-i*
unfolding *vec-to-iarray-morph*[*symmetric*] **unfolding** *row-add-def* **using** *from-nat-ia-not-i*
by *vector*
qed

lemma *matrix-to-iarray-interchange-columns*[*code-unfold*]:
fixes *A*::*'a*::{*semiring-1*}^*columns*::{*mod-type*}^*rows*::{*mod-type*}
shows *matrix-to-iarray* (*interchange-columns A i j*) = *interchange-columns-iarray* (*matrix-to-iarray A*) (*to-nat i*) (*to-nat j*)
unfolding *interchange-columns-def interchange-columns-iarray-def o-def tabulate2-def*
unfolding *nrows-eq-card-rows ncols-eq-card-columns*
unfolding *matrix-to-iarray-def o-def vec-to-iarray-def*
by (*auto simp add*: *to-nat-from-nat-id to-nat-less-card*[*of i*] *to-nat-less-card*[*of j*])

lemma *matrix-to-iarray-mult-columns*[*code-unfold*]:
fixes *A*::*'a*::{*semiring-1*}^*columns*::{*mod-type*}^*rows*::{*mod-type*}
shows *matrix-to-iarray* (*mult-column A i q*) = *mult-column-iarray* (*matrix-to-iarray A*) (*to-nat i*) *q*
unfolding *mult-column-def mult-column-iarray-def o-def tabulate2-def*
unfolding *nrows-eq-card-rows ncols-eq-card-columns*
unfolding *matrix-to-iarray-def o-def vec-to-iarray-def*
by (*auto simp add*: *to-nat-from-nat-id*)

lemma *matrix-to-iarray-column-add*[*code-unfold*]:
fixes *A*::*'a*::{*semiring-1*}^*columns*::{*mod-type*}^*rows*::{*mod-type*}
shows *matrix-to-iarray* (*column-add A i j q*) = *column-add-iarray* (*matrix-to-iarray A*) (*to-nat i*) (*to-nat j*) *q*
unfolding *column-add-def column-add-iarray-def o-def tabulate2-def*
unfolding *nrows-eq-card-rows ncols-eq-card-columns*
unfolding *matrix-to-iarray-def o-def vec-to-iarray-def*
by (*auto simp add*: *to-nat-from-nat-id to-nat-less-card*[*of i*] *to-nat-less-card*[*of j*])

end

17 Gauss Jordan algorithm over nested IArrays

theory *Gauss-Jordan-IArrays*
imports
Matrix-To-IArray
Gauss-Jordan
begin

17.1 Definitions and functions to compute the Gauss-Jordan algorithm over matrices represented as nested iarrays

definition *least-non-zero-position-of-vector-from-index A i* = *the* (*List.find* (λ*x*. *A* !! *x* ≠ *0*) [*i*..<*IArray.length A*])
definition *least-non-zero-position-of-vector A* = *least-non-zero-position-of-vector-from-index A 0*

definition *vector-all-zero-from-index* :: (*nat* × *'a*::{*zero*} *iarray*) => *bool*

where *vector-all-zero-from-index A′ = (let i=fst A′; A=(snd A′) in IArray.all (λx. A!!x = 0) (IArray [i..<(IArray.length A)]))*

definition *Gauss-Jordan-in-ij-iarrays :: 'a::{field} iarray iarray => nat => nat => 'a iarray iarray*
where *Gauss-Jordan-in-ij-iarrays A i j = (let n = least-non-zero-position-of-vector-from-index (column-iarray j A) i;*
interchange-A = interchange-rows-iarray A i n;
A′ = mult-row-iarray interchange-A i (1/interchange-A!!i!!j)
in IArray.of-fun (λs. if s = i then A′ !! s else row-add-iarray A′ s i (− interchange-A !! s !! j) !! s) (nrows-iarray A))

definition *Gauss-Jordan-column-k-iarrays :: (nat × 'a::{field} iarray iarray) => nat => (nat × 'a iarray iarray)*
where *Gauss-Jordan-column-k-iarrays A′ k=(let A=(snd A′); i=(fst A′) in*
if ((vector-all-zero-from-index (i, (column-iarray k A)))) ∨ i = (nrows-iarray A) then (i,A) else (Suc i, (Gauss-Jordan-in-ij-iarrays A i k)))

definition *Gauss-Jordan-upt-k-iarrays :: 'a::{field} iarray iarray => nat => 'a::{field} iarray iarray*
where *Gauss-Jordan-upt-k-iarrays A k = snd (foldl Gauss-Jordan-column-k-iarrays (0,A) [0..<Suc k])*

definition *Gauss-Jordan-iarrays :: 'a::{field} iarray iarray => 'a::{field} iarray iarray*
where *Gauss-Jordan-iarrays A = Gauss-Jordan-upt-k-iarrays A (ncols-iarray A − 1)*

17.2 Proving the equivalence between Gauss-Jordan algorithm over nested iarrays and over nested vecs (abstract matrices).

lemma *vector-all-zero-from-index-eq*:
fixes *A::'a::{zero}^'n::{mod-type}*
shows *(∀ m≥i. A $ m = 0) = (vector-all-zero-from-index (to-nat i, vec-to-iarray A))*
proof *(auto simp add: vector-all-zero-from-index-def Let-def Option.is-none-def find-None-iff)*
fix *x*
assume *zero*: *∀ m≥i. A $ m = 0*
and *x-length*: *x<length (IArray.list-of (vec-to-iarray A))* **and** *i-le-x*: *to-nat i ≤ x*
have *x-le-card*: *x < CARD('n)* **using** *x-length* **unfolding** *vec-to-iarray-def* **by** *auto*
have *i-le-from-nat-x*: *i ≤ from-nat x* **using** *from-nat-mono'[OF i-le-x x-le-card]* **unfolding** *from-nat-to-nat-id* **.**
hence *Axk*: *A $ (from-nat x) = 0* **using** *zero* **by** *simp*
have *vec-to-iarray A !! x = vec-to-iarray A !! to-nat (from-nat x::'n)* **unfolding** *to-nat-from-nat-id[OF x-le-card]* **..**

also have ... = *A* $ (*from-nat x*) **unfolding** *vec-to-iarray-nth'* **..**
also have ... = *0* **unfolding** *Axk* **..**
finally show *IArray.list-of* (*vec-to-iarray A*) ! *x* = *0*
unfolding *IArray.sub-def* **.**
next
fix *m*::'*n*
assume *zero-assm*: ∀ *x*∈{*mod-type-class.to-nat i*..<*length* (*IArray.list-of* (*vec-to-iarray A*))}. *IArray.list-of* (*vec-to-iarray A*) ! *x* = *0*
and *i-le-m*: *i* ≤ *m*
have *zero*: ∀ *x*<*length* (*IArray.list-of* (*vec-to-iarray A*)). *mod-type-class.to-nat i* ≤ *x* ⟶ *IArray.list-of* (*vec-to-iarray A*) ! *x* = *0*
using *zero-assm* **by** *auto*
have *to-nat-i-le-m*:*to-nat i* ≤ *to-nat m* **using** *to-nat-mono'*[*OF i-le-m*] **.**
have *m-le-length*: *to-nat m* < *IArray.length* (*vec-to-iarray A*) **unfolding** *vec-to-iarray-def* **using** *to-nat-less-card* **by** *auto*
have *A* $ *m* = *vec-to-iarray A* !! (*to-nat m*) **unfolding** *vec-to-iarray-nth'* **..**
also have ... = *0* **using** *zero to-nat-i-le-m m-le-length* **unfolding** *nrows-iarray-def* **by** (*metis IArray.sub-def IArray.length-def*)
finally show *A* $ *m* = *0* **.**
qed

lemma *matrix-vector-all-zero-from-index*:
fixes *A*::'*a*::{*zero*} ^'*columns*::{*mod-type*} ^'*rows*::{*mod-type*}
shows (∀ *m*≥*i*. *A* $ *m* $ *k* = *0*) = (*vector-all-zero-from-index* (*to-nat i*, *vec-to-iarray* (*column k A*)))
unfolding *vector-all-zero-from-index-eq*[*symmetric*] *column-def* **by** *simp*

lemma *vec-to-iarray-least-non-zero-position-of-vector-from-index*:
fixes *A*::'*a*::{*zero*} ^'*n*::{*mod-type*}
assumes *not-all-zero*: ¬ (*vector-all-zero-from-index* (*to-nat i*, *vec-to-iarray A*))
shows *least-non-zero-position-of-vector-from-index* (*vec-to-iarray A*) (*to-nat i*) = *to-nat* (*LEAST n*. *A* $ *n* ≠ *0* ∧ *i* ≤ *n*)
proof −
have ∃ *a*. *List.find* (λ*x*. *vec-to-iarray A* !! *x* ≠ *0*) [*to-nat i*..<*IArray.length* (*vec-to-iarray A*)] = *Some a*
proof (*rule ccontr*, *simp*, *unfold IArray.sub-def*[*symmetric*] *IArray.length-def*[*symmetric*])
assume *List.find* (λ*x*. (*vec-to-iarray A*) !! *x* ≠ *0*) [*to-nat i*..<*IArray.length* (*vec-to-iarray A*)] = *None*
hence ¬ (∃ *x*. *x* ∈ *set* [*mod-type-class.to-nat i*..<*IArray.length* (*vec-to-iarray A*)] ∧ *vec-to-iarray A* !! *x* ≠ *0*)
unfolding *find-None-iff* **.**
thus *False* **using** *not-all-zero* **unfolding** *vector-all-zero-from-index-eq*[*symmetric*]
by (*simp del*: *IArray.length-def IArray.sub-def*, *unfold length-vec-to-iarray*, *metis to-nat-less-card to-nat-mono' vec-to-iarray-nth'*)
qed
from *this* **obtain** *a* **where** *a*: *List.find* (λ*x*. *vec-to-iarray A* !! *x* ≠ *0*) [*to-nat i*..<*IArray.length* (*vec-to-iarray A*)] = *Some a*
by *blast*

from *this* **obtain** *ia* **where**
ia-less-length: *ia*<*length* [*to-nat i*..<*IArray.length* (*vec-to-iarray A*)] **and**
not-eq-zero: *vec-to-iarray A* !! ([*to-nat i*..<*IArray.length* (*vec-to-iarray A*)] ! *ia*) ≠ *0* **and**
a-eq: *a* = [*to-nat i*..<*IArray.length* (*vec-to-iarray A*)] ! *ia*
and *least*: (∀ *ja*<*ia*. ¬ *vec-to-iarray A* !! ([*to-nat i*..<*IArray.length* (*vec-to-iarray A*)] ! *ja*) ≠ *0*)
unfolding *find-Some-iff* **by** *blast*
have *not-eq-zero'*: *vec-to-iarray A* !! *a* ≠ *0* **using** *not-eq-zero* **unfolding** *a-eq* **.**
have *i-less-a*: *to-nat i* ≤ *a* **using** *ia-less-length length-upt nth-upt a-eq* **by** *auto*
have *a-less-card*: *a*<*CARD*('*n*) **using** *a-eq ia-less-length* **unfolding** *vec-to-iarray-def* **by** *auto*
have (*LEAST n. A* $ *n* ≠ *0* ∧ *i* ≤ *n*) = *from-nat a*
proof (*rule Least-equality*, *rule conjI*)
show *A* $ *from-nat a* ≠ *0* **unfolding** *vec-to-iarray-nth'*[*symmetric*] **using** *not-eq-zero'* **unfolding** *to-nat-from-nat-id*[*OF a-less-card*] **.**
show *i* ≤ *from-nat a* **using** *a-less-card from-nat-mono' from-nat-to-nat-id i-less-a* **by** *fastforce*
fix *x* **assume** *A* $ *x* ≠ *0* ∧ *i* ≤ *x* **hence** *Axj*: *A* $ *x* ≠ *0* **and** *i-le-x*: *i* ≤ *x* **by** *fast+*
show *from-nat a* ≤ *x*
proof (*rule ccontr*)
assume ¬ *from-nat a* ≤ *x* **hence** *x-less-from-nat-a*: *x* < *from-nat a* **by** *simp*
define *ja* **where** *ja* = (*to-nat x*) − (*to-nat i*)
have *to-nat-x-less-card*: *to-nat x* < *CARD* ('*n*) **using** *bij-to-nat*[**where** *?'a*='*n*] **unfolding** *bij-betw-def* **by** *fastforce*
hence *ja-less-length*: *ja* < *IArray.length* (*vec-to-iarray A*) **unfolding** *ja-def vec-to-iarray-def* **by** *auto*
have [*to-nat i*..<*IArray.length* (*vec-to-iarray A*)] ! *ja* = *to-nat i* + *ja*
by (*rule nth-upt*, *unfold vec-to-iarray-def*,*auto*, *metis add-diff-inverse diff-add-zero ja-def not-less-iff-gr-or-eq to-nat-less-card*)
also have *i-plus-ja*: ... = *to-nat x* **unfolding** *ja-def* **by** (*simp add*: *i-le-x to-nat-mono'*)
finally have *list-rw*: [*to-nat i*..<*IArray.length* (*vec-to-iarray A*)] ! *ja* = *to-nat x* **.**
moreover have *ja*<*ia*
proof −
have *a* = *to-nat i* + *ia* **unfolding** *a-eq*
by (*rule nth-upt*, *metis ia-less-length length-upt less-diff-conv add.commute*)
thus *?thesis* **by** (*metis i-plus-ja add-less-cancel-right add.commute to-nat-le x-less-from-nat-a*)
qed
ultimately have *vec-to-iarray A* !! (*to-nat x*) = *0* **using** *least* **by** *auto*
hence *A* $ *x* = *0* **unfolding** *vec-to-iarray-nth'* **.**
thus *False* **using** *Axj* **by** *contradiction*
qed
qed
hence *a* = *to-nat* (*LEAST n. A* $ *n* ≠ *0* ∧ *i* ≤ *n*) **using** *to-nat-from-nat-id*[*OF a-less-card*] **by** *simp*

thus *?thesis* **unfolding** *least-non-zero-position-of-vector-from-index-def* **unfolding** *a* **by** *simp*
qed

corollary *vec-to-iarray-least-non-zero-position-of-vector-from-index'*:
fixes *A*::*'a*::{*zero*}^*cols*::{*mod-type*}^*rows*::{*mod-type*}
assumes *not-all-zero*: ¬ (*vector-all-zero-from-index* (*to-nat i*, *vec-to-iarray* (*column j A*)))
shows *least-non-zero-position-of-vector-from-index* (*vec-to-iarray* (*column j A*)) (*to-nat i*) = *to-nat* (*LEAST n*. *A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*)
unfolding *vec-to-iarray-least-non-zero-position-of-vector-from-index*[*OF not-all-zero*]
unfolding *column-def* **by** *fastforce*

corollary *vec-to-iarray-least-non-zero-position-of-vector-from-index''*:
fixes *A*::*'a*::{*zero*}^*cols*::{*mod-type*}^*rows*::{*mod-type*}
assumes *not-all-zero*: ¬ (*vector-all-zero-from-index* (*to-nat j*, *vec-to-iarray* (*row i A*)))
shows *least-non-zero-position-of-vector-from-index* (*vec-to-iarray* (*row i A*)) (*to-nat j*) = *to-nat* (*LEAST n*. *A* $ *i* $ *n* ≠ *0* ∧ *j* ≤ *n*)
unfolding *vec-to-iarray-least-non-zero-position-of-vector-from-index*[*OF not-all-zero*]
unfolding *row-def* **by** *fastforce*

lemma *matrix-to-iarray-Gauss-Jordan-in-ij*[*code-unfold*]:
fixes *A*::*'a*::{*field*}^*columns*::{*mod-type*}^*rows*::{*mod-type*}
assumes *not-all-zero*: ¬ (*vector-all-zero-from-index* (*to-nat i*, *vec-to-iarray* (*column j A*)))
shows *matrix-to-iarray* (*Gauss-Jordan-in-ij A i j*) = *Gauss-Jordan-in-ij-iarrays* (*matrix-to-iarray A*) (*to-nat i*) (*to-nat j*)
proof (*unfold Gauss-Jordan-in-ij-def Gauss-Jordan-in-ij-iarrays-def Let-def*, *rule matrix-to-iarray-eq-of-fun*, *auto simp del*: *IArray.sub-def IArray.length-def*)
show *vec-to-iarray* (*mult-row* (*interchange-rows A i* (*LEAST n*. *A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*)) *i* (*1* / *A* $ (*LEAST n*. *A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) $ *j*) $ *i*) =
mult-row-iarray
(*interchange-rows-iarray* (*matrix-to-iarray A*) (*to-nat i*)
(*least-non-zero-position-of-vector-from-index* (*column-iarray* (*to-nat j*) (*matrix-to-iarray A*)) (*to-nat i*)))
(*to-nat i*) (*1* / *interchange-rows-iarray* (*matrix-to-iarray A*) (*to-nat i*)
(*least-non-zero-position-of-vector-from-index* (*column-iarray* (*to-nat j*) (*matrix-to-iarray A*)) (*to-nat i*)) !! *to-nat i* !! *to-nat j*) !! *to-nat i*
unfolding *vec-to-iarray-column*[*symmetric*]
unfolding *vec-to-iarray-least-non-zero-position-of-vector-from-index'*[*OF not-all-zero*]
unfolding *matrix-to-iarray-interchange-rows*[*symmetric*]
unfolding *matrix-to-iarray-mult-row*[*symmetric*]
unfolding *matrix-to-iarray-nth*
unfolding *interchange-rows-i*
unfolding *vec-matrix* **..**
next

fix *ia*
show *vec-to-iarray*
(*row-add* (*mult-row* (*interchange-rows A i* (*LEAST n. A $ n $ j* ≠ *0* ∧ *i* ≤ *n*)) *i* (*1* / *A $* (*LEAST n. A $ n $ j* ≠ *0* ∧ *i* ≤ *n*) *$ j*)) *ia i*
(− *interchange-rows A i* (*LEAST n. A $ n $ j* ≠ *0* ∧ *i* ≤ *n*) *$ ia $ j*) *$ ia*) =
row-add-iarray
(*mult-row-iarray*
(*interchange-rows-iarray* (*matrix-to-iarray A*) (*to-nat i*)
(*least-non-zero-position-of-vector-from-index* (*column-iarray* (*to-nat j*) (*matrix-to-iarray A*)) (*to-nat i*)))
(*to-nat i*)
(*1* / *interchange-rows-iarray* (*matrix-to-iarray A*) (*to-nat i*)
(*least-non-zero-position-of-vector-from-index* (*column-iarray* (*to-nat j*) (*matrix-to-iarray A*)) (*to-nat i*)) !!
to-nat i !! *to-nat j*))
(*to-nat ia*) (*to-nat i*)
(− *interchange-rows-iarray* (*matrix-to-iarray A*) (*to-nat i*)
(*least-non-zero-position-of-vector-from-index* (*column-iarray* (*to-nat j*) (*matrix-to-iarray A*)) (*to-nat i*)) !!
to-nat ia !! *to-nat j*) !! *to-nat ia*
unfolding *vec-to-iarray-column*[*symmetric*]
unfolding *vec-to-iarray-least-non-zero-position-of-vector-from-index'*[*OF not-all-zero*]
unfolding *matrix-to-iarray-interchange-rows*[*symmetric*]
unfolding *matrix-to-iarray-mult-row*[*symmetric*]
unfolding *matrix-to-iarray-nth*
unfolding *interchange-rows-i*
unfolding *matrix-to-iarray-row-add*[*symmetric*]
unfolding *vec-matrix* **..**
next
show *nrows-iarray* (*matrix-to-iarray A*) =
IArray.length (*matrix-to-iarray*
(χ *s. if s = i then mult-row* (*interchange-rows A i* (*LEAST n. A $ n $ j* ≠ *0* ∧ *i* ≤ *n*)) *i* (*1* / *interchange-rows A i* (*LEAST n. A $ n $ j* ≠ *0* ∧ *i* ≤ *n*) *$ i $ j*) *$ s*
else row-add (*mult-row* (*interchange-rows A i* (*LEAST n. A $ n $ j* ≠ *0* ∧ *i* ≤ *n*)) *i* (*1* / *interchange-rows A i* (*LEAST n. A $ n $ j* ≠ *0* ∧ *i* ≤ *n*) *$ i $ j*)) *s i*
(− *interchange-rows A i* (*LEAST n. A $ n $ j* ≠ *0* ∧ *i* ≤ *n*) *$ s $ j*) *$ s*))
unfolding *length-eq-card-rows nrows-eq-card-rows* **..**
qed

lemma *matrix-to-iarray-Gauss-Jordan-column-k-1*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *k*: *k*<*ncols A*
and *i*: *i*≤*nrows A*
shows (*fst* (*Gauss-Jordan-column-k* (*i*, *A*) *k*)) = *fst* (*Gauss-Jordan-column-k-iarrays* (*i*, *matrix-to-iarray A*) *k*)

proof (*cases i<nrows A*)
case *True*
show *?thesis*
unfolding *Gauss-Jordan-column-k-def Let-def Gauss-Jordan-column-k-iarrays-def fst-conv snd-conv*
unfolding *vec-to-iarray-column*[*of from-nat k A*, *unfolded to-nat-from-nat-id*[*OF k*[*unfolded ncols-def*]], *symmetric*]
using *matrix-vector-all-zero-from-index*[*symmetric*, *of from-nat i::'rows from-nat k::'columns*]
unfolding *to-nat-from-nat-id*[*OF True*[*unfolded nrows-def*]] *to-nat-from-nat-id*[*OF k*[*unfolded ncols-def*]]
using *matrix-to-iarray-Gauss-Jordan-in-ij*
unfolding *matrix-to-iarray-nrows snd-conv* **by** *auto*
next
case *False*
have *vector-all-zero-from-index* (*nrows A*, *column-iarray k* (*matrix-to-iarray A*))
unfolding *vector-all-zero-from-index-def* **unfolding** *Let-def snd-conv fst-conv*
unfolding *nrows-def column-iarray-def*
unfolding *length-eq-card-rows* **by** (*simp add*: *is-none-code*(*1*))
thus *?thesis*
using *i False*
unfolding *Gauss-Jordan-column-k-iarrays-def Gauss-Jordan-column-k-def Let-def*
by *auto*
qed

lemma *matrix-to-iarray-Gauss-Jordan-column-k-2*:
fixes *A::'a::{field}^'columns::{mod-type}^'rows::{mod-type}*
assumes *k*: *k<ncols A*
and *i*: *i≤nrows A*
shows *matrix-to-iarray* (*snd* (*Gauss-Jordan-column-k* (*i*, *A*) *k*)) = *snd* (*Gauss-Jordan-column-k-iarrays* (*i*, *matrix-to-iarray A*) *k*)
proof (*cases i<nrows A*)
case *True* **show** *?thesis*
unfolding *Gauss-Jordan-column-k-def Let-def Gauss-Jordan-column-k-iarrays-def fst-conv snd-conv*
unfolding *vec-to-iarray-column*[*of from-nat k A*, *unfolded to-nat-from-nat-id*[*OF k*[*unfolded ncols-def*]], *symmetric*]
unfolding *matrix-vector-all-zero-from-index*[*symmetric*, *of from-nat i::'rows from-nat k::'columns*, *symmetric*]
using *matrix-to-iarray-Gauss-Jordan-in-ij*[*of from-nat i::'rows from-nat k::'columns*]

unfolding *to-nat-from-nat-id*[*OF True*[*unfolded nrows-def*]] *to-nat-from-nat-id*[*OF k*[*unfolded ncols-def*]]
unfolding *matrix-to-iarray-nrows* **by** *auto*
next
case *False* **show** *?thesis*
using *assms False* **unfolding** *Gauss-Jordan-column-k-def Let-def Gauss-Jordan-column-k-iarrays-def*
by (*auto simp add*: *matrix-to-iarray-nrows*)
qed

Due to the assumptions presented in ⟦*?k* < *ncols ?A*; *?i* ≤ *nrows ?A*⟧ ⟹ *matrix-to-iarray* (*snd* (*Gauss-Jordan-column-k* (*?i*, *?A*) *?k*)) = *snd* (*Gauss-Jordan-column-k-iarrays* (*?i*, *matrix-to-iarray ?A*) *?k*), the following lemma must have three shows. The proof style is similar to *?k* < *ncols ?A* ⟹ *reduced-row-echelon-form-upt-k* (*Gauss-Jordan-upt-k ?A ?k*) (*Suc ?k*)

?k < *ncols ?A* ⟹ *foldl Gauss-Jordan-column-k* (*0*, *?A*) [*0*..<*Suc ?k*] = (*if* ∀ *m*. *is-zero-row-upt-k m* (*Suc ?k*) (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *?A*) [*0*..<*Suc ?k*])) *then 0 else mod-type-class.to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n* (*Suc ?k*) (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *?A*) [*0*..<*Suc ?k*]))) + *1*, *snd* (*foldl Gauss-Jordan-column-k* (*0*, *?A*) [*0*..<*Suc ?k*])).

lemma *foldl-Gauss-Jordan-column-k-eq*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *k*: *k*<*ncols A*
shows *matrix-to-iarray-Gauss-Jordan-upt-k*[*code-unfold*]: *matrix-to-iarray* (*Gauss-Jordan-upt-k A k*) = *Gauss-Jordan-upt-k-iarrays* (*matrix-to-iarray A*) *k*
and *fst-foldl-Gauss-Jordan-column-k-eq*: *fst* (*foldl Gauss-Jordan-column-k-iarrays* (*0*, *matrix-to-iarray A*) [*0*..<*Suc k*]) = *fst* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc k*])
and *fst-foldl-Gauss-Jordan-column-k-less*: *fst* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc k*]) ≤ *nrows A*
using *assms*
proof (*induct k*)
show *matrix-to-iarray* (*Gauss-Jordan-upt-k A 0*) = *Gauss-Jordan-upt-k-iarrays* (*matrix-to-iarray A*) *0*
unfolding *Gauss-Jordan-upt-k-def Gauss-Jordan-upt-k-iarrays-def* **by** (*auto*, *metis k le0 less-nat-zero-code matrix-to-iarray-Gauss-Jordan-column-k-2 neq0-conv*)

show *fst* (*foldl Gauss-Jordan-column-k-iarrays* (*0*, *matrix-to-iarray A*) [*0*..<*Suc 0*]) = *fst* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc 0*])
unfolding *Gauss-Jordan-upt-k-def Gauss-Jordan-upt-k-iarrays-def* **by** (*auto*, *metis gr-implies-not0 k le0 matrix-to-iarray-Gauss-Jordan-column-k-1 neq0-conv*)
show *fst* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc 0*]) ≤ *nrows A* **unfolding** *Gauss-Jordan-upt-k-def* **by** (*simp add*: *Gauss-Jordan-column-k-def Let-def size1 nrows-def*)
next
fix *k*
assume (*k* < *ncols A* ⟹ *matrix-to-iarray* (*Gauss-Jordan-upt-k A k*) = *Gauss-Jordan-upt-k-iarrays* (*matrix-to-iarray A*) *k*) **and**
(*k* < *ncols A* ⟹ *fst* (*foldl Gauss-Jordan-column-k-iarrays* (*0*, *matrix-to-iarray A*) [*0*..<*Suc k*]) = *fst* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc k*]))
and (*k* < *ncols A* ⟹ *fst* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc k*]) ≤ *nrows A*)
and *Suc-k-less-card*: *Suc k* < *ncols A*
hence *hyp1*: *matrix-to-iarray* (*Gauss-Jordan-upt-k A k*) = *Gauss-Jordan-upt-k-iarrays* (*matrix-to-iarray A*) *k*

and *hyp2*: *fst* (*foldl Gauss-Jordan-column-k-iarrays* (*0*, *matrix-to-iarray A*) [*0..<Suc k*]) = *fst* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0..<Suc k*])
and *hyp3*: *fst* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0..<Suc k*]) ≤ *nrows A*
by *auto*
hence *hyp1-unfolded*: *matrix-to-iarray* (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0..<Suc k*])) = *snd* (*foldl Gauss-Jordan-column-k-iarrays* (*0*, *matrix-to-iarray A*) [*0..<Suc k*])
using *hyp1* **unfolding** *Gauss-Jordan-upt-k-def Gauss-Jordan-upt-k-iarrays-def*
by *simp*
have *upt-rw*: [*0..<Suc* (*Suc k*)] = [*0..<Suc k*] @ [(*Suc k*)] **by** *auto*
have *fold-rw*: (*foldl Gauss-Jordan-column-k-iarrays* (*0*, *matrix-to-iarray A*) [*0..<Suc k*])
= (*fst* (*foldl Gauss-Jordan-column-k-iarrays* (*0*, *matrix-to-iarray A*) [*0..<Suc k*]), *snd* (*foldl Gauss-Jordan-column-k-iarrays* (*0*, *matrix-to-iarray A*) [*0..<Suc k*]))
by *simp*
have *fold-rw'*: (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0..<*(*Suc k*)])
= (*fst* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0..<*(*Suc k*)]), *snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0..<*(*Suc k*)])) **by** *simp*
show *fst* (*foldl Gauss-Jordan-column-k-iarrays* (*0*, *matrix-to-iarray A*) [*0..<Suc* (*Suc k*)]) = *fst* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0..<Suc* (*Suc k*)])
unfolding *upt-rw foldl-append* **unfolding** *List.foldl.simps* **apply** (*subst fold-rw*)
apply (*subst fold-rw'*) **unfolding** *hyp2* **unfolding** *hyp1-unfolded*[*symmetric*]
proof (*rule matrix-to-iarray-Gauss-Jordan-column-k-1*[*symmetric*, *of Suc k* (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0..<Suc k*]))])
show *Suc k < ncols* (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0..<Suc k*]))
using *Suc-k-less-card* **unfolding** *ncols-def* **.**
show *fst* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0..<Suc k*]) ≤ *nrows* (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0..<Suc k*])) **using** *hyp3* **unfolding** *nrows-def* **.**
qed
show *matrix-to-iarray* (*Gauss-Jordan-upt-k A* (*Suc k*)) = *Gauss-Jordan-upt-k-iarrays* (*matrix-to-iarray A*) (*Suc k*)
unfolding *Gauss-Jordan-upt-k-def Gauss-Jordan-upt-k-iarrays-def upt-rw foldl-append List.foldl.simps*
apply (*subst fold-rw*) **apply** (*subst fold-rw'*) **unfolding** *hyp2 hyp1-unfolded*[*symmetric*]
proof (*rule matrix-to-iarray-Gauss-Jordan-column-k-2*, *unfold ncols-def nrows-def*)
show *Suc k < CARD*('*columns*) **using** *Suc-k-less-card* **unfolding** *ncols-def* **.**
show *fst* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0..<Suc k*]) ≤ *CARD*('*rows*)
using *hyp3* **unfolding** *nrows-def* **.**
qed
show *fst* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0..<Suc* (*Suc k*)]) ≤ *nrows A*
using [[*unfold-abs-def* = *false*]]
unfolding *upt-rw foldl-append* **unfolding** *List.foldl.simps* **apply** (*subst fold-rw'*)
unfolding *Gauss-Jordan-column-k-def Let-def*
using *hyp3 le-antisym not-less-eq-eq* **unfolding** *nrows-def* **by** *fastforce*
qed

lemma *matrix-to-iarray-Gauss-Jordan*[*code-unfold*]:

fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*}
shows *matrix-to-iarray* (*Gauss-Jordan A*) = *Gauss-Jordan-iarrays* (*matrix-to-iarray A*)
unfolding *Gauss-Jordan-iarrays-def ncols-iarray-def* **unfolding** *length-eq-card-columns*
by (*auto simp add*: *Gauss-Jordan-def matrix-to-iarray-Gauss-Jordan-upt-k ncols-def*)

17.3 Implementation over IArrays of the computation of the *rank* of a matrix

definition *rank-iarray* :: *'a*::{*field*} *iarray iarray* => *nat*
where *rank-iarray A* = (*let A'* = (*Gauss-Jordan-iarrays A*); *nrows* = (*IArray.length A'*) *in card* {*i*. *i*<*nrows* ∧ ¬ *is-zero-iarray* (*A'* !! *i*)})

17.3.1 Proving the equivalence between *rank* and *rank-iarray*.

First of all, some code equations are removed to allow the execution of Gauss-Jordan algorithm using iarrays

lemmas *card'-code*(*2*)[*code del*]
lemmas *rank-Gauss-Jordan-code*[*code del*]

lemma *rank-eq-card-iarrays*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*}
shows *rank A* = *card* {*vec-to-iarray* (*row i* (*Gauss-Jordan A*)) |*i*. ¬ *is-zero-iarray* (*vec-to-iarray* (*row i* (*Gauss-Jordan A*)))}
proof (*unfold rank-Gauss-Jordan-eq Let-def*, *rule bij-betw-same-card*[*of vec-to-iarray*], *auto simp add*: *bij-betw-def*)
show *inj-on vec-to-iarray* {*row i* (*Gauss-Jordan A*) |*i*. *row i* (*Gauss-Jordan A*) ≠ *0*} **using** *inj-vec-to-iarray* **unfolding** *inj-on-def* **by** *blast*
fix *i* **assume** *r*: *row i* (*Gauss-Jordan A*) ≠ *0*
show ∃ *ia*. *vec-to-iarray* (*row i* (*Gauss-Jordan A*)) = *vec-to-iarray* (*row ia* (*Gauss-Jordan A*)) ∧ ¬ *is-zero-iarray* (*vec-to-iarray* (*row ia* (*Gauss-Jordan A*)))
proof (*rule exI*[*of* - *i*], *simp*)
show ¬ *is-zero-iarray* (*vec-to-iarray* (*row i* (*Gauss-Jordan A*))) **using** *r* **unfolding** *is-zero-iarray-eq-iff* **.**
qed
next
fix *i*
assume *not-zero-iarray*: ¬ *is-zero-iarray* (*vec-to-iarray* (*row i* (*Gauss-Jordan A*)))
show *vec-to-iarray* (*row i* (*Gauss-Jordan A*)) ∈ *vec-to-iarray* ‘ {*row i* (*Gauss-Jordan A*) |*i*. *row i* (*Gauss-Jordan A*) ≠ *0*}
by (*rule imageI*, *auto simp add*: *not-zero-iarray is-zero-iarray-eq-iff*)
qed

lemma *rank-eq-card-iarrays'*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*}

shows *rank A* = (*let A'* = (*Gauss-Jordan-iarrays* (*matrix-to-iarray A*)) *in card* {*row-iarray* (*to-nat i*) *A'* |*i*::*'rows*. ¬ *is-zero-iarray* (*A'* !! (*to-nat i*))})
unfolding *Let-def* **unfolding** *rank-eq-card-iarrays vec-to-iarray-row' matrix-to-iarray-Gauss-Jordan row-iarray-def* **..**

lemma *rank-eq-card-iarrays-code*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*}
shows *rank A* = (*let A'* = (*Gauss-Jordan-iarrays* (*matrix-to-iarray A*)) *in card* {*i*::*'rows*. ¬ *is-zero-iarray* (*A'* !! (*to-nat i*))})
proof (*unfold rank-eq-card-iarrays' Let-def*, *rule bij-betw-same-card*[*symmetric*, *of* λ*i*. *row-iarray* (*to-nat i*) (*Gauss-Jordan-iarrays* (*matrix-to-iarray A*))],
unfold bij-betw-def inj-on-def, *auto*, *unfold IArray.sub-def*[*symmetric*])
fix *x y*::*'rows*
assume *x*: ¬ *is-zero-iarray* (*Gauss-Jordan-iarrays* (*matrix-to-iarray A*) !! *to-nat x*)
and *y*: ¬ *is-zero-iarray* (*Gauss-Jordan-iarrays* (*matrix-to-iarray A*) !! *to-nat y*)
and *eq*: *row-iarray* (*to-nat x*) (*Gauss-Jordan-iarrays* (*matrix-to-iarray A*)) = *row-iarray* (*to-nat y*) (*Gauss-Jordan-iarrays* (*matrix-to-iarray A*))
have *eq'*: (*Gauss-Jordan A*) $ *x* = (*Gauss-Jordan A*) $ *y* **by** (*metis eq matrix-to-iarray-Gauss-Jordan row-iarray-def vec-matrix vec-to-iarray-morph*)
hence *not-zero-x*: ¬ *is-zero-row x* (*Gauss-Jordan A*) **and** *not-zero-y*: ¬ *is-zero-row y* (*Gauss-Jordan A*)
by (*metis is-zero-iarray-eq-iff is-zero-row-def' matrix-to-iarray-Gauss-Jordan vec-eq-iff vec-matrix x zero-index*)+
hence *x-in*: *row x* (*Gauss-Jordan A*) ∈ {*row i* (*Gauss-Jordan A*) |*i*::*'rows*. *row i* (*Gauss-Jordan A*) ≠ *0*}
and *y-in*: *row y* (*Gauss-Jordan A*) ∈ {*row i* (*Gauss-Jordan A*) |*i*::*'rows*. *row i* (*Gauss-Jordan A*) ≠ *0*}
by (*metis* (*lifting*, *mono-tags*) *is-zero-iarray-eq-iff matrix-to-iarray-Gauss-Jordan mem-Collect-eq vec-to-iarray-row' x y*)+
show *x* = *y* **using** *inj-index-independent-rows*[*OF - x-in eq'*] *rref-Gauss-Jordan* **by** *fast*
qed

17.3.2 Code equations for computing the rank over nested iarrays and the dimensions of the elementary subspaces

lemma *rank-iarrays-code*[*code*]:
rank-iarray A = *length* (*filter* (λ*x*. ¬ *is-zero-iarray x*) (*IArray.list-of* (*Gauss-Jordan-iarrays A*)))
proof −
obtain *xs* **where** *A-eq-xs*: (*Gauss-Jordan-iarrays A*) = *IArray xs* **by** (*metis iarray.exhaust*)
have *rank-iarray A* = *card* {*i*. *i*<(*IArray.length* (*Gauss-Jordan-iarrays A*)) ∧ ¬ *is-zero-iarray* ((*Gauss-Jordan-iarrays A*) !! *i*)} **unfolding** *rank-iarray-def Let-def* **..**
also have ... = *length* (*filter* (λ*x*. ¬ *is-zero-iarray x*) (*IArray.list-of* (*Gauss-Jordan-iarrays A*)))
unfolding *A-eq-xs* **using** *length-filter-conv-card*[*symmetric*] **by** *force*

finally show *?thesis* **.**
qed

lemma *matrix-to-iarray-rank*[*code-unfold*]:
shows *rank A* = *rank-iarray* (*matrix-to-iarray A*)
unfolding *rank-eq-card-iarrays-code rank-iarray-def Let-def*
apply (*rule bij-betw-same-card*[*of to-nat*])
unfolding *bij-betw-def*
apply *auto*
unfolding *IArray.length-def*[*symmetric*] *IArray.sub-def*[*symmetric*] **apply** (*metis inj-onI to-nat-eq*)
unfolding *matrix-to-iarray-Gauss-Jordan*[*symmetric*] *length-eq-card-rows*
using *bij-to-nat*[**where** *?'a='c*] **unfolding** *bij-betw-def* **by** *auto*

lemma *dim-null-space-iarray*[*code-unfold*]:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*}
shows *vec.dim* (*null-space A*) = *ncols-iarray* (*matrix-to-iarray A*) − *rank-iarray* (*matrix-to-iarray A*)
unfolding *dim-null-space ncols-eq-card-columns matrix-to-iarray-rank dimension-vector* **by** *simp*

lemma *dim-col-space-iarray*[*code-unfold*]:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*}
shows *vec.dim* (*col-space A*) = *rank-iarray* (*matrix-to-iarray A*)
unfolding *rank-eq-dim-col-space*[*of A, symmetric*] *matrix-to-iarray-rank* **..**

lemma *dim-row-space-iarray*[*code-unfold*]:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*}
shows *vec.dim* (*row-space A*) = *rank-iarray* (*matrix-to-iarray A*)
unfolding *row-rank-def*[*symmetric*] *rank-def*[*symmetric*] *matrix-to-iarray-rank* **..**

lemma *dim-left-null-space-space-iarray*[*code-unfold*]:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*}
shows *vec.dim* (*left-null-space A*) = *nrows-iarray* (*matrix-to-iarray A*) − *rank-iarray* (*matrix-to-iarray A*)
unfolding *dim-left-null-space nrows-eq-card-rows matrix-to-iarray-rank dimension-vector* **by** *auto*

end

18 Obtaining explicitly the invertible matrix which transforms a matrix to its reduced row echelon form over nested iarrays

theory *Gauss-Jordan-PA-IArrays*
imports
Gauss-Jordan-PA
Gauss-Jordan-IArrays

begin

18.1 Definitions

definition *Gauss-Jordan-in-ij-iarrays-PA A′ i j =*
(let P = fst A′; A = snd A′; n = least-non-zero-position-of-vector-from-index (column-iarray j A) i; interchange-A = interchange-rows-iarray A i n;
interchange-P = interchange-rows-iarray P i n; P′ = mult-row-iarray interchange-P i (1 / interchange-A !! i !! j)
in (IArray.of-fun (λs. if s = i then P′ !! s else row-add-iarray P′ s i (− interchange-A !! s !! j) !! s) (nrows-iarray A), Gauss-Jordan-in-ij-iarrays A i j))

definition *Gauss-Jordan-column-k-iarrays-PA :: ('a::{field} iarray iarray × nat × 'a::{field} iarray iarray) => nat => ('a iarray iarray × nat × 'a iarray iarray)*
where *Gauss-Jordan-column-k-iarrays-PA A′ k=(let P = fst A′; i = fst (snd A′); A = snd (snd A′) in*
if (vector-all-zero-from-index (i, (column-iarray k A))) ∨ i = (nrows-iarray A) then (P,i,A) else let Gauss = Gauss-Jordan-in-ij-iarrays-PA (P, A) i k
in (fst Gauss, i + 1, snd Gauss))

definition *Gauss-Jordan-upt-k-iarrays-PA :: 'a::{field} iarray iarray => nat => ('a::{field} iarray iarray × 'a iarray iarray)*
where *Gauss-Jordan-upt-k-iarrays-PA A k = (let foldl = foldl Gauss-Jordan-column-k-iarrays-PA (mat-iarray 1 (nrows-iarray A), 0, A) [0..<Suc k] in (fst foldl, snd (snd foldl)))*

definition *Gauss-Jordan-iarrays-PA :: 'a::{field} iarray iarray => ('a iarray iarray × 'a iarray iarray)*
where *Gauss-Jordan-iarrays-PA A = Gauss-Jordan-upt-k-iarrays-PA A (ncols-iarray A − 1)*

18.2 Proofs

18.2.1 Properties of *Gauss-Jordan-in-ij-iarrays-PA*

lemma *Gauss-Jordan-in-ij-iarrays-PA-def'[code]:*
Gauss-Jordan-in-ij-iarrays-PA A′ i j =
(let P = fst A′; A = snd A′; n = least-non-zero-position-of-vector-from-index (column-iarray j A) i;
interchange-A = interchange-rows-iarray A i n; A′ = mult-row-iarray interchange-A i (1 / interchange-A !! i !! j);
interchange-P = interchange-rows-iarray P i n; P′ = mult-row-iarray interchange-P i (1 / interchange-A !! i !! j)
in (IArray.of-fun (λs. if s = i then P′ !! s else row-add-iarray P′ s i (− interchange-A !! s !! j) !! s) (nrows-iarray A),
(IArray.of-fun (λs. if s = i then A′ !! s else row-add-iarray A′ s i (− interchange-A !! s !! j) !! s) (nrows-iarray A))))
unfolding *Gauss-Jordan-in-ij-iarrays-PA-def Gauss-Jordan-in-ij-iarrays-def Let-def*
..

lemma *snd-Gauss-Jordan-in-ij-iarrays-PA*:
shows *snd* (*Gauss-Jordan-in-ij-iarrays-PA* (*P*, *A*) *i j*) = *Gauss-Jordan-in-ij-iarrays A i j*
unfolding *Gauss-Jordan-in-ij-iarrays-PA-def Gauss-Jordan-in-ij-iarrays-def Let-def snd-conv* **..**

lemma *matrix-to-iarray-snd-Gauss-Jordan-in-ij-iarrays-PA*:
assumes ¬ *vector-all-zero-from-index* (*to-nat i*, *vec-to-iarray* (*column j A*))
shows *matrix-to-iarray* (*snd* (*Gauss-Jordan-in-ij-PA* (*P*,*A*) *i j*)) = *snd* (*Gauss-Jordan-in-ij-iarrays-PA* (*matrix-to-iarray P*,*matrix-to-iarray A*) (*to-nat i*) (*to-nat j*))
by (*metis assms matrix-to-iarray-Gauss-Jordan-in-ij snd-Gauss-Jordan-in-ij-PA-eq snd-Gauss-Jordan-in-ij-iarrays-PA*)

lemma *matrix-to-iarray-fst-Gauss-Jordan-in-ij-iarrays-PA*:
assumes *not-all-zero*: ¬ *vector-all-zero-from-index* (*to-nat i*, *vec-to-iarray* (*column j A*))
shows *matrix-to-iarray* (*fst* (*Gauss-Jordan-in-ij-PA* (*P*,*A*) *i j*)) = *fst* (*Gauss-Jordan-in-ij-iarrays-PA* (*matrix-to-iarray P*,*matrix-to-iarray A*) (*to-nat i*) (*to-nat j*))
proof (*unfold Gauss-Jordan-in-ij-PA-def Gauss-Jordan-in-ij-iarrays-PA-def Let-def fst-conv snd-conv*, *rule matrix-to-iarray-eq-of-fun*, *auto simp del*: *IArray.length-def IArray.sub-def*)
show *vec-to-iarray* (*mult-row* (*interchange-rows P i* (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*)) *i* (*1* / *A* $ (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) $ *j*) $ *i*) =
mult-row-iarray (*interchange-rows-iarray* (*matrix-to-iarray P*) (*to-nat i*)
(*least-non-zero-position-of-vector-from-index* (*column-iarray* (*to-nat j*) (*matrix-to-iarray A*)) (*to-nat i*))) (*to-nat i*)
(*1* / *interchange-rows-iarray* (*matrix-to-iarray A*) (*to-nat i*)
(*least-non-zero-position-of-vector-from-index* (*column-iarray* (*to-nat j*) (*matrix-to-iarray A*)) (*to-nat i*)) !! *to-nat i* !! *to-nat j*) !! *to-nat i*
unfolding *vec-to-iarray-column*[*symmetric*]
unfolding *vec-to-iarray-least-non-zero-position-of-vector-from-index'*[*OF not-all-zero*]
unfolding *matrix-to-iarray-interchange-rows*[*symmetric*]
unfolding *matrix-to-iarray-mult-row*[*symmetric*]
unfolding *matrix-to-iarray-nth*
unfolding *interchange-rows-i*
unfolding *vec-matrix* **..**
next
fix *ia*
show *vec-to-iarray* (*row-add* (*mult-row* (*interchange-rows P i* (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*)) *i* (*1* / *A* $ (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) $ *j*)) *ia i*
(− *interchange-rows A i* (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) $ *ia* $ *j*) $ *ia*) =
row-add-iarray (*mult-row-iarray* (*interchange-rows-iarray* (*matrix-to-iarray P*) (*to-nat i*)
(*least-non-zero-position-of-vector-from-index* (*column-iarray* (*to-nat j*) (*matrix-to-iarray A*)) (*to-nat i*))) (*to-nat i*)
(*1* / *interchange-rows-iarray* (*matrix-to-iarray A*) (*to-nat i*)
(*least-non-zero-position-of-vector-from-index* (*column-iarray* (*to-nat*

j) (*matrix-to-iarray A*)) (*to-nat i*)) !!
to-nat i !! to-nat j)) (to-nat ia) (to-nat i)
(− *interchange-rows-iarray* (*matrix-to-iarray A*) (*to-nat i*)
(*least-non-zero-position-of-vector-from-index* (*column-iarray* (*to-nat j*)
(*matrix-to-iarray A*)) (*to-nat i*)) !!
to-nat ia !! *to-nat j*) !! *to-nat ia*
unfolding *vec-to-iarray-column*[*symmetric*]
unfolding *vec-to-iarray-least-non-zero-position-of-vector-from-index'*[*OF not-all-zero*]
unfolding *matrix-to-iarray-interchange-rows*[*symmetric*]
unfolding *matrix-to-iarray-mult-row*[*symmetric*]
unfolding *matrix-to-iarray-nth*
unfolding *interchange-rows-i*
unfolding *matrix-to-iarray-row-add*[*symmetric*]
unfolding *vec-matrix* **..**
next
show *nrows-iarray* (*matrix-to-iarray A*) = *IArray.length* (*matrix-to-iarray*
(χ *s. if s* = *i then mult-row* (*interchange-rows P i* (*LEAST n. A* $ *n* $ *j* ≠ *0*
∧ *i* ≤ *n*)) *i* (*1 / interchange-rows A i* (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) $ *i* $
j) $ *s*
else row-add (*mult-row* (*interchange-rows P i* (*LEAST n. A* $ *n* $ *j* ≠ *0*
∧ *i* ≤ *n*)) *i* (*1 / interchange-rows A i* (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) $ *i* $
j)) *s i*
(− *interchange-rows A i* (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) $ *s* $ *j*) $
s))
unfolding *length-eq-card-rows nrows-eq-card-rows* **..**
qed

18.2.2 Properties about *Gauss-Jordan-column-k-iarrays-PA*

lemma *matrix-to-iarray-fst-Gauss-Jordan-column-k-PA*:
assumes *i*: *i*≤*nrows A* **and** *k*: *k*<*ncols A*
shows *matrix-to-iarray* (*fst* (*Gauss-Jordan-column-k-PA* (*P*,*i*,*A*) *k*)) = *fst* (*Gauss-Jordan-column-k-iarrays-PA*
(*matrix-to-iarray P*, *i*, *matrix-to-iarray A*) *k*)
proof (*cases i*<*nrows A*)
case *True*
show *?thesis*
unfolding *Gauss-Jordan-column-k-PA-def Gauss-Jordan-column-k-iarrays-PA-def*
Let-def
unfolding *snd-conv fst-conv*
unfolding *matrix-vector-all-zero-from-index*
unfolding *to-nat-from-nat-id*[*OF True*[*unfolded nrows-def*]]
unfolding *matrix-to-iarray-nrows*
using *matrix-to-iarray-fst-Gauss-Jordan-in-ij-iarrays-PA*[*of from-nat i from-nat k*
A P]
using *matrix-to-iarray-snd-Gauss-Jordan-in-ij-iarrays-PA*[*of from-nat i from-nat k*
A P]
unfolding *to-nat-from-nat-id*[*OF True*[*unfolded nrows-def*]]
unfolding *to-nat-from-nat-id*[*OF k*[*unfolded ncols-def*]]
unfolding *vec-to-iarray-column'*[*OF k*] **by** *auto*

next
case *False*
hence *i-eq-nrows*:*i*=*nrows A* **using** *i* **by** *simp*
thus *?thesis*
unfolding *Gauss-Jordan-column-k-PA-def Gauss-Jordan-column-k-iarrays-PA-def Let-def*
unfolding *snd-conv fst-conv* **unfolding** *matrix-to-iarray-nrows* **by** *fastforce*
qed

lemma *matrix-to-iarray-snd-Gauss-Jordan-column-k-PA*:
assumes *i*: *i*≤*nrows A* **and** *k*: *k*<*ncols A*
shows (*fst* (*snd* (*Gauss-Jordan-column-k-PA* (*P*,*i*,*A*) *k*))) = *fst* (*snd* (*Gauss-Jordan-column-k-iarrays-PA* (*matrix-to-iarray P*, *i*, *matrix-to-iarray A*) *k*))
using *matrix-to-iarray-Gauss-Jordan-column-k-1*[*OF k i*]
unfolding *Gauss-Jordan-column-k-def Gauss-Jordan-column-k-iarrays-def*
unfolding *Gauss-Jordan-column-k-PA-def Gauss-Jordan-column-k-iarrays-PA-def snd-conv fst-conv Let-def*
unfolding *snd-Gauss-Jordan-in-ij-iarrays-PA*
unfolding *snd-if-conv*
unfolding *snd-Gauss-Jordan-in-ij-PA-eq*
by *fastforce*

lemma *matrix-to-iarray-third-Gauss-Jordan-column-k-PA*:
assumes *i*: *i*≤*nrows A* **and** *k*: *k*<*ncols A*
shows *matrix-to-iarray* (*snd* (*snd* (*Gauss-Jordan-column-k-PA* (*P*,*i*,*A*) *k*))) = *snd* (*snd* (*Gauss-Jordan-column-k-iarrays-PA* (*matrix-to-iarray P*, *i*, *matrix-to-iarray A*) *k*))
using *matrix-to-iarray-Gauss-Jordan-column-k-2*[*OF k i*]
unfolding *Gauss-Jordan-column-k-def Gauss-Jordan-column-k-iarrays-def*
unfolding *Gauss-Jordan-column-k-PA-def Gauss-Jordan-column-k-iarrays-PA-def snd-conv fst-conv Let-def*
unfolding *snd-Gauss-Jordan-in-ij-iarrays-PA*
unfolding *snd-if-conv snd-Gauss-Jordan-in-ij-PA-eq* **by** *fast*

18.2.3 Properties about *Gauss-Jordan-upt-k-iarrays-PA*

lemma
assumes *k*<*ncols A*
shows *matrix-to-iarray-fst-Gauss-Jordan-upt-k-PA*: *matrix-to-iarray* (*fst* (*Gauss-Jordan-upt-k-PA A k*)) = *fst* (*Gauss-Jordan-upt-k-iarrays-PA* (*matrix-to-iarray A*) *k*)
and *matrix-to-iarray-snd-Gauss-Jordan-upt-k-PA*: *matrix-to-iarray* (*snd* (*Gauss-Jordan-upt-k-PA A k*)) = (*snd* (*Gauss-Jordan-upt-k-iarrays-PA* (*matrix-to-iarray A*) *k*))
and *fst* (*snd* (*foldl Gauss-Jordan-column-k-PA* (*mat 1*, *0*, *A*) [*0*..<*Suc k*])) =
  fst (*snd* (*foldl Gauss-Jordan-column-k-iarrays-PA* (*mat-iarray 1* (*nrows-iarray* (*matrix-to-iarray A*)), *0*, *matrix-to-iarray A*) [*0*..<*Suc k*]))
and *fst* (*snd* (*foldl Gauss-Jordan-column-k-PA* (*mat 1*, *0*, *A*) [*0*..<*Suc k*])) ≤ *nrows A*
using *assms*

proof (*induct k*)
assume *zero-less-ncols*: *0 < ncols A*
show *matrix-to-iarray* (*fst* (*Gauss-Jordan-upt-k-PA A 0*)) = *fst* (*Gauss-Jordan-upt-k-iarrays-PA* (*matrix-to-iarray A*) *0*)
unfolding *Gauss-Jordan-upt-k-PA-def Gauss-Jordan-upt-k-iarrays-PA-def Let-def fst-conv* **apply** *auto*
unfolding *nrows-eq-card-rows* **unfolding** *matrix-to-iarray-mat*[*symmetric*]
by (*rule matrix-to-iarray-fst-Gauss-Jordan-column-k-PA*, *auto simp add*: *zero-less-ncols*)
show *matrix-to-iarray* (*snd* (*Gauss-Jordan-upt-k-PA A 0*)) = *snd* (*Gauss-Jordan-upt-k-iarrays-PA* (*matrix-to-iarray A*) *0*)
unfolding *Gauss-Jordan-upt-k-PA-def Gauss-Jordan-upt-k-iarrays-PA-def Let-def fst-conv snd-conv* **apply** *auto*
unfolding *nrows-eq-card-rows* **unfolding** *matrix-to-iarray-mat*[*symmetric*]
by (*rule matrix-to-iarray-third-Gauss-Jordan-column-k-PA*, *auto simp add*: *zero-less-ncols*)
show *fst* (*snd* (*foldl Gauss-Jordan-column-k-PA* (*mat 1*, *0*, *A*) [*0..<Suc 0*])) =
fst (*snd* (*foldl Gauss-Jordan-column-k-iarrays-PA* (*mat-iarray 1* (*nrows-iarray* (*matrix-to-iarray A*)), *0*, *matrix-to-iarray A*) [*0..<Suc 0*]))
apply *simp* **unfolding** *nrows-eq-card-rows* **unfolding** *matrix-to-iarray-mat*[*symmetric*]
by (*rule matrix-to-iarray-snd-Gauss-Jordan-column-k-PA*, *auto simp add*: *zero-less-ncols*)
show *fst* (*snd* (*foldl Gauss-Jordan-column-k-PA* (*mat 1*, *0*, *A*) [*0..<Suc 0*])) ≤ *nrows A*
by (*simp add*: *fst-snd-Gauss-Jordan-column-k-PA-eq fst-Gauss-Jordan-column-k*)
next
fix *k* **assume** (*k < ncols A* ⟹ *matrix-to-iarray* (*fst* (*Gauss-Jordan-upt-k-PA A k*)) = *fst* (*Gauss-Jordan-upt-k-iarrays-PA* (*matrix-to-iarray A*) *k*))
and (*k < ncols A* ⟹ *matrix-to-iarray* (*snd* (*Gauss-Jordan-upt-k-PA A k*)) = *snd* (*Gauss-Jordan-upt-k-iarrays-PA* (*matrix-to-iarray A*) *k*))
and (*k < ncols A* ⟹ *fst* (*snd* (*foldl Gauss-Jordan-column-k-PA* (*mat 1*, *0*, *A*) [*0..<Suc k*])) =
fst (*snd* (*foldl Gauss-Jordan-column-k-iarrays-PA* (*mat-iarray 1* (*nrows-iarray* (*matrix-to-iarray A*)), *0*, *matrix-to-iarray A*) [*0..<Suc k*])))
and (*k < ncols A* ⟹ *fst* (*snd* (*foldl Gauss-Jordan-column-k-PA* (*mat 1*, *0*, *A*) [*0..<Suc k*])) ≤ *nrows A*)
and *Suc-k*: *Suc k < ncols A*
hence *hyp1*: *matrix-to-iarray* (*fst* (*Gauss-Jordan-upt-k-PA A k*)) = *fst* (*Gauss-Jordan-upt-k-iarrays-PA* (*matrix-to-iarray A*) *k*)
and *hyp2*: *matrix-to-iarray* (*snd* (*Gauss-Jordan-upt-k-PA A k*)) = *snd* (*Gauss-Jordan-upt-k-iarrays-PA* (*matrix-to-iarray A*) *k*)
and *hyp3*: *fst* (*snd* (*foldl Gauss-Jordan-column-k-PA* (*mat 1*, *0*, *A*) [*0..<Suc k*]))
=
fst (*snd* (*foldl Gauss-Jordan-column-k-iarrays-PA* (*mat-iarray 1* (*nrows-iarray* (*matrix-to-iarray A*)), *0*, *matrix-to-iarray A*) [*0..<Suc k*]))
and *hyp4*: *fst* (*snd* (*foldl Gauss-Jordan-column-k-PA* (*mat 1*, *0*, *A*) [*0..<Suc k*]))
≤ *nrows A*
by *linarith*+

have *suc-rw*: [*0..<Suc* (*Suc k*)] = [*0..<Suc k*] @ [*Suc k*] **by** *simp*
define *A'* **where** *A'* = *foldl Gauss-Jordan-column-k-PA* (*mat 1*, *0*, *A*) [*0..<Suc k*]

define B **where** $B = foldl\ Gauss\text{-}Jordan\text{-}column\text{-}k\text{-}iarrays\text{-}PA\ (mat\text{-}iarray\ 1\ (nrows\text{-}iarray\ (matrix\text{-}to\text{-}iarray\ A)),\ 0,\ matrix\text{-}to\text{-}iarray\ A)\ [0..<Suc\ k]$
have $A'\text{-}eq$: $A' = (fst\ A',\ fst\ (snd\ A'),\ snd\ (snd\ A'))$ **by** $auto$

have $fst\text{-}A'$: $matrix\text{-}to\text{-}iarray\ (fst\ A') = fst\ B$ **unfolding** $A'\text{-}def\ B\text{-}def$ **by** $(rule\ hyp1[unfolded\ Gauss\text{-}Jordan\text{-}upt\text{-}k\text{-}PA\text{-}def\ Gauss\text{-}Jordan\text{-}upt\text{-}k\text{-}iarrays\text{-}PA\text{-}def\ Let\text{-}def\ fst\text{-}conv])$
have $fst\text{-}snd\text{-}A'$: $fst\ (snd\ A') = fst\ (snd\ B)$ **unfolding** $A'\text{-}def\ B\text{-}def$ **by** $(rule\ hyp3[unfolded\ Gauss\text{-}Jordan\text{-}upt\text{-}k\text{-}PA\text{-}def\ Gauss\text{-}Jordan\text{-}upt\text{-}k\text{-}iarrays\text{-}PA\text{-}def\ ])$
have $snd\text{-}snd\text{-}A'$: $matrix\text{-}to\text{-}iarray\ (snd\ (snd\ A')) = (snd\ (snd\ B))$
unfolding $A'\text{-}def\ B\text{-}def$
by $(rule\ hyp2[unfolded\ Gauss\text{-}Jordan\text{-}upt\text{-}k\text{-}PA\text{-}def\ Gauss\text{-}Jordan\text{-}upt\text{-}k\text{-}iarrays\text{-}PA\text{-}def\ Let\text{-}def\ fst\text{-}conv\ snd\text{-}conv])$

show $matrix\text{-}to\text{-}iarray\ (fst\ (Gauss\text{-}Jordan\text{-}upt\text{-}k\text{-}PA\ A\ (Suc\ k))) = fst\ (Gauss\text{-}Jordan\text{-}upt\text{-}k\text{-}iarrays\text{-}PA\ (matrix\text{-}to\text{-}iarray\ A)\ (Suc\ k))$
proof −
have $matrix\text{-}to\text{-}iarray\ (fst\ (Gauss\text{-}Jordan\text{-}upt\text{-}k\text{-}PA\ A\ (Suc\ k))) = matrix\text{-}to\text{-}iarray\ (fst\ (foldl\ Gauss\text{-}Jordan\text{-}column\text{-}k\text{-}PA\ (mat\ 1,\ 0,\ A)\ [0..<Suc\ (Suc\ k)]))$
unfolding $Gauss\text{-}Jordan\text{-}upt\text{-}k\text{-}PA\text{-}def\ Let\text{-}def\ fst\text{-}conv$ **..**
also have $... = matrix\text{-}to\text{-}iarray\ (fst\ (Gauss\text{-}Jordan\text{-}column\text{-}k\text{-}PA\ (foldl\ Gauss\text{-}Jordan\text{-}column\text{-}k\text{-}PA\ (mat\ 1,\ 0,\ A)\ [0..<Suc\ k])\ (Suc\ k)))$
unfolding $suc\text{-}rw\ foldl\text{-}append$ **unfolding** $List.foldl.simps$ **..**
also have $... = matrix\text{-}to\text{-}iarray\ (fst\ (Gauss\text{-}Jordan\text{-}column\text{-}k\text{-}PA\ (fst\ A', fst\ (snd\ A'),\ snd\ (snd\ A'))\ (Suc\ k)))$ **unfolding** $A'\text{-}def$ **by** $simp$
also have $... = fst\ (Gauss\text{-}Jordan\text{-}column\text{-}k\text{-}iarrays\text{-}PA\ (matrix\text{-}to\text{-}iarray\ (fst\ A'),\ fst\ (snd\ A'),\ matrix\text{-}to\text{-}iarray\ (snd\ (snd\ A')))\ (Suc\ k))$
proof $(rule\ matrix\text{-}to\text{-}iarray\text{-}fst\text{-}Gauss\text{-}Jordan\text{-}column\text{-}k\text{-}PA)$
show $fst\ (snd\ A') \leq nrows\ (snd\ (snd\ A'))$ **using** $hyp4$ **unfolding** $nrows\text{-}def\ A'\text{-}def$
.
show $Suc\ k < ncols\ (snd\ (snd\ A'))$ **using** $Suc\text{-}k$ **unfolding** $ncols\text{-}def$.
qed
also have $... = fst\ (Gauss\text{-}Jordan\text{-}column\text{-}k\text{-}iarrays\text{-}PA\ (fst\ B,\ fst\ (snd\ B),\ snd\ (snd\ B))\ (Suc\ k))$
unfolding $fst\text{-}A'\ fst\text{-}snd\text{-}A'\ snd\text{-}snd\text{-}A'$ **..**
also have $... = fst\ (Gauss\text{-}Jordan\text{-}upt\text{-}k\text{-}iarrays\text{-}PA\ (matrix\text{-}to\text{-}iarray\ A)\ (Suc\ k))$
unfolding $Gauss\text{-}Jordan\text{-}upt\text{-}k\text{-}iarrays\text{-}PA\text{-}def\ Let\text{-}def\ fst\text{-}conv$
unfolding $suc\text{-}rw\ foldl\text{-}append$ **unfolding** $List.foldl.simps\ B\text{-}def$ **by** $fastforce$
finally show $?thesis$.
qed

show $matrix\text{-}to\text{-}iarray\ (snd\ (Gauss\text{-}Jordan\text{-}upt\text{-}k\text{-}PA\ A\ (Suc\ k))) = snd\ (Gauss\text{-}Jordan\text{-}upt\text{-}k\text{-}iarrays\text{-}PA\ (matrix\text{-}to\text{-}iarray\ A)\ (Suc\ k))$
proof −
have $matrix\text{-}to\text{-}iarray\ (snd\ (Gauss\text{-}Jordan\text{-}upt\text{-}k\text{-}PA\ A\ (Suc\ k))) = matrix\text{-}to\text{-}iarray\ (snd\ (snd\ (foldl\ Gauss\text{-}Jordan\text{-}column\text{-}k\text{-}PA\ (mat\ 1,\ 0,\ A)\ [0..<Suc\ (Suc\ k)])))$
unfolding $Gauss\text{-}Jordan\text{-}upt\text{-}k\text{-}PA\text{-}def\ Let\text{-}def\ snd\text{-}conv$ **..**
also have $... = matrix\text{-}to\text{-}iarray\ (snd\ (snd\ (Gauss\text{-}Jordan\text{-}column\text{-}k\text{-}PA\ (foldl\ Gauss\text{-}Jordan\text{-}column\text{-}k\text{-}PA\ (mat\ 1,\ 0,\ A)\ [0..<Suc\ k])\ (Suc\ k))))$

unfolding *suc-rw foldl-append* **unfolding** *List.foldl.simps* **..**
also have ... = *matrix-to-iarray* (*snd* (*snd* (*Gauss-Jordan-column-k-PA* (*fst A'*,*fst* (*snd A'*), *snd* (*snd A'*)) (*Suc k*)))) **unfolding** *A'-def* **by** *simp*
also have ... = *snd* (*snd* (*Gauss-Jordan-column-k-iarrays-PA* (*matrix-to-iarray* (*fst A'*), *fst* (*snd A'*), *matrix-to-iarray* (*snd* (*snd A'*))) (*Suc k*)))
proof (*rule matrix-to-iarray-third-Gauss-Jordan-column-k-PA*)
show *fst* (*snd A'*) ≤ *nrows* (*snd* (*snd A'*)) **using** *hyp4* **unfolding** *nrows-def A'-def*
.
show *Suc k* < *ncols* (*snd* (*snd A'*)) **using** *Suc-k* **unfolding** *ncols-def* **.**
qed
also have ... = *snd* (*snd* (*Gauss-Jordan-column-k-iarrays-PA* (*fst B*, *fst* (*snd B*), *snd* (*snd B*)) (*Suc k*)))
unfolding *fst-A' fst-snd-A' snd-snd-A'* **..**
also have ... = *snd* (*Gauss-Jordan-upt-k-iarrays-PA* (*matrix-to-iarray A*) (*Suc k*))

unfolding *Gauss-Jordan-upt-k-iarrays-PA-def Let-def fst-conv*
unfolding *suc-rw foldl-append* **unfolding** *List.foldl.simps B-def* **by** *fastforce*
finally show *?thesis* **.**
qed

show *fst* (*snd* (*foldl Gauss-Jordan-column-k-PA* (*mat 1*, *0*, *A*) [*0*..<*Suc* (*Suc k*)])) =
fst (*snd* (*foldl Gauss-Jordan-column-k-iarrays-PA* (*mat-iarray 1* (*nrows-iarray* (*matrix-to-iarray A*)), *0*, *matrix-to-iarray A*) [*0*..<*Suc* (*Suc k*)]))
proof −
have *fst* (*snd* (*foldl Gauss-Jordan-column-k-PA* (*mat 1*, *0*, *A*) [*0*..<*Suc* (*Suc k*)])) =
fst (*snd* (*Gauss-Jordan-column-k-PA* (*foldl Gauss-Jordan-column-k-PA* (*mat 1*, *0*, *A*) [*0*..<*Suc k*]) (*Suc k*)))
unfolding *suc-rw foldl-append* **unfolding** *List.foldl.simps* **..**
also have ... = *fst* (*snd* (*Gauss-Jordan-column-k-PA* (*fst A'*,*fst* (*snd A'*), *snd* (*snd A'*)) (*Suc k*)))
unfolding *A'-def* **by** *simp*
also have ... = *fst* (*snd* (*Gauss-Jordan-column-k-iarrays-PA* (*matrix-to-iarray* (*fst A'*),*fst* (*snd A'*), *matrix-to-iarray* (*snd* (*snd A'*))) (*Suc k*)))
proof (*rule matrix-to-iarray-snd-Gauss-Jordan-column-k-PA*)
show *fst* (*snd A'*) ≤ *nrows* (*snd* (*snd A'*)) **using** *hyp4* **unfolding** *nrows-def A'-def*
.
show *Suc k* < *ncols* (*snd* (*snd A'*)) **using** *Suc-k* **unfolding** *ncols-def* **.**
qed
also have ... = *fst* (*snd* (*Gauss-Jordan-column-k-iarrays-PA* (*fst B*,*fst* (*snd B*), *snd* (*snd B*)) (*Suc k*)))
unfolding *fst-A' fst-snd-A' snd-snd-A'* **..**
also have ... = *fst* (*snd* (*foldl Gauss-Jordan-column-k-iarrays-PA* (*mat-iarray 1* (*nrows-iarray* (*matrix-to-iarray A*)), *0*, *matrix-to-iarray A*) [*0*..<*Suc* (*Suc k*)]))
unfolding *B-def* **unfolding** *nrows-eq-card-rows* **unfolding** *matrix-to-iarray-mat*[*symmetric*]
by *auto*
finally show *?thesis* **.**
qed

show *fst* (*snd* (*foldl Gauss-Jordan-column-k-PA* (*mat 1*, *0*, *A*) [*0*..<*Suc* (*Suc k*)]))
≤ *nrows A*
by (*metis snd-foldl-Gauss-Jordan-column-k-eq Suc-k fst-foldl-Gauss-Jordan-column-k-less*)
qed

18.2.4 Properties about *Gauss-Jordan-iarrays-PA*

lemma *matrix-to-iarray-fst-Gauss-Jordan-PA*:
shows *matrix-to-iarray* (*fst* (*Gauss-Jordan-PA A*)) = *fst* (*Gauss-Jordan-iarrays-PA* (*matrix-to-iarray A*))
unfolding *Gauss-Jordan-PA-def Gauss-Jordan-iarrays-PA-def*
using *matrix-to-iarray-fst-Gauss-Jordan-upt-k-PA*[*of ncols A − 1 A*]
by (*simp add*: *ncols-def ncols-eq-card-columns*)

lemma *matrix-to-iarray-snd-Gauss-Jordan-PA*:
shows *matrix-to-iarray* (*snd* (*Gauss-Jordan-PA A*)) = *snd* (*Gauss-Jordan-iarrays-PA* (*matrix-to-iarray A*))
unfolding *Gauss-Jordan-PA-def Gauss-Jordan-iarrays-PA-def*
using *matrix-to-iarray-snd-Gauss-Jordan-upt-k-PA*[*of ncols A − 1 A*]
by (*simp add*: *ncols-def ncols-eq-card-columns*)

lemma *Gauss-Jordan-iarrays-PA-mult*:
fixes *A*::'*a*::{*field*}^'*cols*::{*mod-type*}^'*rows*::{*mod-type*}
shows *snd* (*Gauss-Jordan-iarrays-PA* (*matrix-to-iarray A*)) = *fst* (*Gauss-Jordan-iarrays-PA* (*matrix-to-iarray A*)) ∗∗*i* (*matrix-to-iarray A*)
proof −
have *snd* (*Gauss-Jordan-PA A*) = *fst* (*Gauss-Jordan-PA A*) ∗∗ *A* **using** *fst-Gauss-Jordan-PA*[*of A*] **..**
hence *matrix-to-iarray* (*snd* (*Gauss-Jordan-PA A*)) = *matrix-to-iarray* (*fst* (*Gauss-Jordan-PA A*) ∗∗ *A*) **by** *simp*
thus *?thesis* **unfolding** *matrix-to-iarray-snd-Gauss-Jordan-PA matrix-to-iarray-matrix-matrix-mult matrix-to-iarray-fst-Gauss-Jordan-PA* **.**
qed

lemma *snd-snd-Gauss-Jordan-column-k-iarrays-PA-eq*:
shows *snd* (*snd* (*Gauss-Jordan-column-k-iarrays-PA* (*P*,*i*,*A*) *k*)) = *snd* (*Gauss-Jordan-column-k-iarrays* (*i*,*A*) *k*)
unfolding *Gauss-Jordan-column-k-iarrays-PA-def Gauss-Jordan-column-k-iarrays-def*
unfolding *Let-def snd-conv fst-conv* **unfolding** *snd-Gauss-Jordan-in-ij-iarrays-PA*
by *auto*

lemma *fst-snd-Gauss-Jordan-column-k-iarrays-PA-eq*:
shows *fst* (*snd* (*Gauss-Jordan-column-k-iarrays-PA* (*P*,*i*,*A*) *k*)) = *fst* (*Gauss-Jordan-column-k-iarrays* (*i*,*A*) *k*)
unfolding *Gauss-Jordan-column-k-iarrays-PA-def Gauss-Jordan-column-k-iarrays-def*

unfolding *Let-def snd-conv fst-conv* **by** *auto*

lemma *foldl-Gauss-Jordan-column-k-iarrays-eq*:
snd (*foldl Gauss-Jordan-column-k-iarrays-PA* (*B*, *0*, *A*) [*0*..<*k*]) = *foldl Gauss-Jordan-column-k-iarrays* (*0*, *A*) [*0*..<*k*]
proof (*induct k*)
case *0*
show *?case* **by** *simp*
case (*Suc k*)
have *suc-rw*: [*0*..<*Suc k*] = [*0*..<*k*] @ [*k*] **by** *simp*
show *?case*
unfolding *suc-rw foldl-append* **unfolding** *List.foldl.simps*
by (*metis Suc.hyps fst-snd-Gauss-Jordan-column-k-iarrays-PA-eq snd-snd-Gauss-Jordan-column-k-iarrays-PA-surjective-pairing*)
qed

lemma *snd-Gauss-Jordan-upt-k-iarrays-PA*:
shows *snd* (*Gauss-Jordan-upt-k-iarrays-PA A k*) = (*Gauss-Jordan-upt-k-iarrays A k*)
unfolding *Gauss-Jordan-upt-k-iarrays-PA-def Gauss-Jordan-upt-k-iarrays-def Let-def*
using *foldl-Gauss-Jordan-column-k-iarrays-eq*[*of mat-iarray 1* (*nrows-iarray A*) *A Suc k*] **by** *simp*

lemma *snd-Gauss-Jordan-iarrays-PA-eq*: *snd* (*Gauss-Jordan-iarrays-PA A*) = *Gauss-Jordan-iarrays A*
unfolding *Gauss-Jordan-iarrays-def Gauss-Jordan-iarrays-PA-def*
using *snd-Gauss-Jordan-upt-k-iarrays-PA* **by** *auto*

end

19 Bases of the four fundamental subspaces over IArrays

theory *Bases-Of-Fundamental-Subspaces-IArrays*
imports
Bases-Of-Fundamental-Subspaces
Gauss-Jordan-PA-IArrays
begin

19.1 Computation of bases of the fundamental subspaces using IArrays

We have made the definitions as efficient as possible.

definition *basis-left-null-space-iarrays A*
= (*let GJ* = *Gauss-Jordan-iarrays-PA A*;
rank-A = *length* [*x*←*IArray.list-of* (*snd GJ*) . ¬ *is-zero-iarray x*]

in set (*map* (λ*i*. *row-iarray i* (*fst GJ*)) [(*rank-A*)..<(*nrows-iarray A*)]))
definition *basis-null-space-iarrays A*
= (*let GJ*= *Gauss-Jordan-iarrays-PA* (*transpose-iarray A*);
rank-A = *length* [*x*←*IArray.list-of* (*snd GJ*) . ¬ *is-zero-iarray x*]
in set (*map* (λ*i*. *row-iarray i* (*fst GJ*)) [(*rank-A*)..<(*ncols-iarray A*)]))
definition *basis-row-space-iarrays A* =
(*let GJ*= *Gauss-Jordan-iarrays A*;
rank-A = *length* [*x*←*IArray.list-of* (*GJ*) . ¬ *is-zero-iarray x*]
in set (*map* (λ*i*. *row-iarray i* (*GJ*)) [*0*..<*rank-A*]))
definition *basis-col-space-iarrays A* = *basis-row-space-iarrays* (*transpose-iarray A*)

The following lemmas make easier the proofs of equivalence between abstract versions and concrete versions. They are false if we remove *matrix-to-iarray*

lemma *basis-null-space-iarrays-eq*:
fixes *A*::*'a*::{*field*}^*cols*::{*mod-type*}^*rows*::{*mod-type*}
shows *basis-null-space-iarrays* (*matrix-to-iarray A*)
= *set* (*map* (λ*i*. *row-iarray i* (*fst* (*Gauss-Jordan-iarrays-PA* (*transpose-iarray* (*matrix-to-iarray A*))))) [(*rank-iarray* (*matrix-to-iarray A*))..<(*ncols-iarray* (*matrix-to-iarray A*))])
unfolding *basis-null-space-iarrays-def Let-def*
unfolding *matrix-to-iarray-rank*[*symmetric*, *of A*]
unfolding *rank-transpose*[*symmetric*, *of A*]
unfolding *matrix-to-iarray-rank*
unfolding *rank-iarrays-code*
unfolding *matrix-to-iarray-transpose*[*symmetric*]
unfolding *snd-Gauss-Jordan-iarrays-PA-eq* **..**

lemma *basis-row-space-iarrays-eq*:
fixes *A*::*'a*::{*field*}^*cols*::{*mod-type*}^*rows*::{*mod-type*}
shows *basis-row-space-iarrays* (*matrix-to-iarray A*) = *set* (*map* (λ*i*. *row-iarray i* (*Gauss-Jordan-iarrays* (*matrix-to-iarray A*))) [*0*..<(*rank-iarray* (*matrix-to-iarray A*))])
unfolding *basis-row-space-iarrays-def Let-def rank-iarrays-code* **..**

lemma *basis-left-null-space-iarrays-eq*:
fixes *A*::*'a*::{*field*}^*cols*::{*mod-type*}^*rows*::{*mod-type*}
shows *basis-left-null-space-iarrays* (*matrix-to-iarray A*) = *basis-null-space-iarrays* (*transpose-iarray* (*matrix-to-iarray A*))
unfolding *basis-left-null-space-iarrays-def*
unfolding *basis-null-space-iarrays-def Let-def*
unfolding *matrix-to-iarray-transpose*[*symmetric*]
unfolding *transpose-transpose*
unfolding *matrix-to-iarray-ncols*[*symmetric*]
unfolding *ncols-transpose*
unfolding *matrix-to-iarray-nrows* **..**

19.2 Code equations

lemma *vec-to-iarray-basis-null-space*[*code-unfold*]:

fixes A::$'a$::{*field*}^$'$*cols*::{*mod-type*}^$'$*rows*::{*mod-type*}
shows *vec-to-iarray'* (*basis-null-space A*) = *basis-null-space-iarrays* (*matrix-to-iarray A*)
proof (*unfold basis-null-space-def basis-null-space-iarrays-eq*, *auto*, *unfold image-def*, *auto*)
fix i::$'$*cols*
assume *rank-le-i*: *rank A* ≤ *to-nat i*
show ∃ *x*∈{*rank-iarray* (*matrix-to-iarray A*)..<*ncols-iarray* (*matrix-to-iarray A*)}. *vec-to-iarray* (*row i* (*P-Gauss-Jordan* (*transpose A*))) = *row-iarray x* (*fst* (*Gauss-Jordan-iarrays-PA* (*transpose-iarray* (*matrix-to-iarray A*))))
proof (*rule bexI*[*of - to-nat i*])
show *to-nat i* ∈ {*rank-iarray* (*matrix-to-iarray A*)..<*ncols-iarray* (*matrix-to-iarray A*)}
unfolding *matrix-to-iarray-ncols*[*symmetric*]
using *rank-le-i to-nat-less-card*[*of i*]
unfolding *matrix-to-iarray-rank ncols-def* **by** *fastforce*
show *vec-to-iarray* (*row i* (*P-Gauss-Jordan* (*transpose A*)))
= *row-iarray* (*to-nat i*) (*fst* (*Gauss-Jordan-iarrays-PA* (*transpose-iarray* (*matrix-to-iarray A*))))
unfolding *matrix-to-iarray-transpose*[*symmetric*]
unfolding *matrix-to-iarray-fst-Gauss-Jordan-PA*[*symmetric*]
unfolding *P-Gauss-Jordan-def*
unfolding *vec-to-iarray-row* **..**
qed
next
fix *i* **assume** *rank-le-i*: *rank-iarray* (*matrix-to-iarray A*) ≤ *i*
and *i-less-nrows*: *i* < *ncols-iarray* (*matrix-to-iarray A*)
hence *i-less-card*:*i* < *CARD* (*'cols*)
unfolding *matrix-to-iarray-ncols*[*symmetric*] *ncols-def* **by** *simp*
show ∃ *x*. (∃ *i*. *x* = *row i* (*P-Gauss-Jordan* (*transpose A*)) ∧ *rank A* ≤ *to-nat i*) ∧ *row-iarray i* (*fst* (*Gauss-Jordan-iarrays-PA* (*transpose-iarray* (*matrix-to-iarray A*)))) = *vec-to-iarray x*
proof (*rule exI*[*of - row* (*from-nat i*) (*P-Gauss-Jordan* (*transpose A*))], *rule conjI*)
show ∃ *ia*. *row* (*from-nat i*) (*P-Gauss-Jordan* (*transpose A*)) = *row ia* (*P-Gauss-Jordan* (*transpose A*)) ∧ *rank A* ≤ *to-nat ia*
by (*rule exI*[*of - from-nat i*],*simp add*: *rank-le-i to-nat-from-nat-id*[*OF i-less-card*] *matrix-to-iarray-rank*)
show *row-iarray i* (*fst* (*Gauss-Jordan-iarrays-PA* (*transpose-iarray* (*matrix-to-iarray A*)))) =
vec-to-iarray (*row* (*from-nat i*) (*P-Gauss-Jordan* (*transpose A*)))
unfolding *matrix-to-iarray-transpose*[*symmetric*]
unfolding *matrix-to-iarray-fst-Gauss-Jordan-PA*[*symmetric*]
unfolding *P-Gauss-Jordan-def*
unfolding *vec-to-iarray-row to-nat-from-nat-id*[*OF i-less-card*] **..**
qed
qed

corollary *vec-to-iarray-basis-left-null-space*[*code-unfold*]:
fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
shows *vec-to-iarray'* (*basis-left-null-space A*) = *basis-left-null-space-iarrays* (*matrix-to-iarray A*)
proof −
have *rw*: *basis-left-null-space A* = *basis-null-space* (*transpose A*)
by (*metis transpose-transpose basis-null-space-eq-basis-left-null-space-transpose*)
show *?thesis* **unfolding** *rw* **unfolding** *basis-left-null-space-iarrays-eq*
using *vec-to-iarray-basis-null-space*[*of transpose A*]
unfolding *matrix-to-iarray-transpose*[*symmetric*] **.**
qed

lemma *vec-to-iarray-basis-row-space*[*code-unfold*]:
fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
shows *vec-to-iarray'* (*basis-row-space A*) = *basis-row-space-iarrays* (*matrix-to-iarray A*)
proof (*unfold basis-row-space-def basis-row-space-iarrays-eq*, *auto*, *unfold image-def*, *auto*)
fix *i*
assume *i*: *row i* (*Gauss-Jordan A*) ≠ *0*
show ∃ *x*∈{*0*..<*rank-iarray* (*matrix-to-iarray A*)}. *vec-to-iarray* (*row i* (*Gauss-Jordan A*)) = *row-iarray x* (*Gauss-Jordan-iarrays* (*matrix-to-iarray A*))
proof (*rule bexI*[*of - to-nat i*])
show *vec-to-iarray* (*row i* (*Gauss-Jordan A*)) = *row-iarray* (*to-nat i*) (*Gauss-Jordan-iarrays* (*matrix-to-iarray A*))
unfolding *vec-to-iarray-row matrix-to-iarray-Gauss-Jordan* **..**
show *to-nat i* ∈ {*0*..<*rank-iarray* (*matrix-to-iarray A*)}
by (*auto*, *unfold matrix-to-iarray-rank*[*symmetric*],
metis (*full-types*) *i iarray-to-vec-vec-to-iarray not-less rank-less-row-i-imp-i-is-zero row-iarray-def vec-matrix vec-to-iarray-row*)
qed
next
fix *i*
assume *i*: *i* < *rank-iarray* (*matrix-to-iarray A*)
hence *i-less-rank*: *i* < *rank A* **unfolding** *matrix-to-iarray-rank* **.**
show ∃ *x*. (∃ *i*. *x* = *row i* (*Gauss-Jordan A*) ∧ *row i* (*Gauss-Jordan A*) ≠ *0*) ∧ *row-iarray i* (*Gauss-Jordan-iarrays* (*matrix-to-iarray A*)) = *vec-to-iarray x*
proof (*rule exI*[*of - row* (*from-nat i*) (*Gauss-Jordan A*)], *rule conjI*)
have *not-zero-i*: ¬ *is-zero-row* (*from-nat i*) (*Gauss-Jordan A*)
proof (*unfold is-zero-row-def*, *rule greatest-ge-nonzero-row'*)
show *reduced-row-echelon-form-upt-k* (*Gauss-Jordan A*) (*ncols* (*Gauss-Jordan A*))
by (*metis rref-Gauss-Jordan rref-implies-rref-upt*)
have *A-not-0*: *A* ≠ *0* **using** *i-less-rank* **by** (*metis less-nat-zero-code rank-0*)
hence *Gauss-not-0*: *Gauss-Jordan A* ≠ *0* **by** (*metis Gauss-Jordan-not-0*)
have *i* ≤ *to-nat* (*GREATEST a*. ¬ *is-zero-row a* (*Gauss-Jordan A*)) **using** *i-less-rank*
unfolding *rank-eq-suc-to-nat-greatest*[*OF A-not-0*] **by** *auto*
thus *from-nat i* ≤ (*GREATEST m*. ¬ *is-zero-row-upt-k m* (*ncols* (*Gauss-Jordan A*)) (*Gauss-Jordan A*)) **unfolding** *is-zero-row-def*[*symmetric*] **by** (*metis leD not-le-imp-less*

to-nat-le)
show ¬ (∀ *a*. *is-zero-row-upt-k a* (*ncols* (*Gauss-Jordan A*)) (*Gauss-Jordan A*)) **using** *Gauss-not-0* **unfolding** *is-zero-row-def*[*symmetric*] *is-zero-row-def'* **by** (*metis vec-eq-iff zero-index*)
qed
have *i-less-card*: *i*<*CARD*('*rows*) **using** *i-less-rank rank-le-nrows*[*of A*] **unfolding** *nrows-def* **by** *simp*
show ∃ *ia*. *row* (*from-nat i*) (*Gauss-Jordan A*) = *row ia* (*Gauss-Jordan A*) ∧ *row ia* (*Gauss-Jordan A*) ≠ *0*
apply (*rule exI*[*of* - *from-nat i*], *simp*) **using** *not-zero-i* **unfolding** *row-def is-zero-row-def' vec-nth-inverse* **by** *auto*
show *row-iarray i* (*Gauss-Jordan-iarrays* (*matrix-to-iarray A*)) = *vec-to-iarray* (*row* (*from-nat i*) (*Gauss-Jordan A*))
unfolding *matrix-to-iarray-Gauss-Jordan*[*symmetric*] *vec-to-iarray-row to-nat-from-nat-id*[*OF i-less-card*] **by** *rule*
qed
qed

corollary *vec-to-iarray-basis-col-space*[*code-unfold*]:
fixes *A*::'*a*::{*field*}^'*cols*::{*mod-type*}^'*rows*::{*mod-type*}
shows *vec-to-iarray*‘ (*basis-col-space A*) = *basis-col-space-iarrays* (*matrix-to-iarray A*)
unfolding *basis-col-space-eq-basis-row-space-transpose basis-col-space-iarrays-def*
unfolding *matrix-to-iarray-transpose*[*symmetric*]
unfolding *vec-to-iarray-basis-row-space* **..**

end

20 Solving systems of equations using the Gauss Jordan algorithm over nested IArrays

theory *System-Of-Equations-IArrays*
imports
System-Of-Equations
Bases-Of-Fundamental-Subspaces-IArrays
begin

20.1 Previous definitions and properties

definition *greatest-not-zero* :: '*a*::{*zero*} *iarray* => *nat*
where *greatest-not-zero A* = *the* (*List.find* (λ*n*. *A* !! *n* ≠ *0*) (*rev* [*0*..<*IArray.length A*]))

lemma *vec-to-iarray-exists*:
shows (∃ *b*. *A* $ *b* ≠ *0*) = *IArray.exists* (λ*b*. (*vec-to-iarray A*) !! *b* ≠ *0*) (*IArray*[*0*..<*IArray.length* (*vec-to-iarray A*)])
proof (*unfold IArray.exists-def length-vec-to-iarray*, *auto simp del*: *IArray.sub-def*)

fix b **assume** Ab: $A \$ b \neq 0$
show $\exists\, b{\in}\{0..{<}CARD('a)\}.\ vec\text{-}to\text{-}iarray\ A\ !!\ b \neq 0$
by (*rule bexI*[*of* - *to-nat b*], *unfold vec-to-iarray-nth'*, *auto simp add*: *Ab to-nat-less-card*[*of b*])
next
fix b **assume** b: $b < CARD('a)$ **and** $Ab\text{-}vec$: $vec\text{-}to\text{-}iarray\ A\ !!\ b \neq 0$
show $\exists\, b.\ A \$ b \neq 0$ **by** (*rule exI*[*of* - *from-nat b*], *metis Ab-vec vec-to-iarray-nth*[*OF b*])
qed

corollary *vec-to-iarray-exists'*:
shows $(\exists\, b.\ A \$ b \neq 0) = IArray.exists\ (\lambda b.\ (vec\text{-}to\text{-}iarray\ A)\ !!\ b \neq 0)\ (IArray\ (rev\ [0..{<}IArray.length\ (vec\text{-}to\text{-}iarray\ A)]))$
by (*simp add*: *vec-to-iarray-exists Option.is-none-def find-None-iff*)

lemma *not-is-zero-iarray-eq-iff*: $(\exists\, b.\ A \$ b \neq 0) = (\neg\ is\text{-}zero\text{-}iarray\ (vec\text{-}to\text{-}iarray\ A))$
by (*metis* (*full-types*) *is-zero-iarray-eq-iff vec-eq-iff zero-index*)

lemma *vec-to-iarray-greatest-not-zero*:
assumes *ex-b*: $(\exists\, b.\ A \$ b \neq 0)$
shows $greatest\text{-}not\text{-}zero\ (vec\text{-}to\text{-}iarray\ A) = to\text{-}nat\ (GREATEST\ b.\ A \$ b \neq 0)$
proof −
let $?P{=}(\lambda n.\ (vec\text{-}to\text{-}iarray\ A)\ !!\ n \neq 0)$
let $?xs{=}(rev\ [0..{<}IArray.length\ (vec\text{-}to\text{-}iarray\ A)])$
have $\exists\, a.\ (List.find\ ?P\ ?xs) = Some\ a$
proof(*rule ccontr*, *simp*, *unfold find-None-iff*)
assume $\neg\ (\exists\, x.\ x \in set\ (rev\ [0..{<}length\ (IArray.list\text{-}of\ (vec\text{-}to\text{-}iarray\ A))]) \wedge IArray.list\text{-}of\ (vec\text{-}to\text{-}iarray\ A)\ !\ x \neq 0)$
thus *False* **using** *ex-b*
unfolding *set-rev* **by** (*auto*, *unfold IArray.length-def*[*symmetric*] *IArray.sub-def*[*symmetric*] *length-vec-to-iarray*,*metis to-nat-less-card vec-to-iarray-nth'*)
qed
from *this* **obtain** a **where** a: $(List.find\ ?P\ ?xs) = Some\ a$ **by** *blast*
from *this* **obtain** *ia* **where** *ia-less-length*: $ia{<}length\ ?xs$
and *P-xs-ia*: $?P\ (?xs!ia)$ **and** *a-eq*: $a = ?xs!ia$ **and** *all-zero*: $(\forall\, j{<}ia.\ \neg\ ?P\ (?xs!j))$
unfolding *find-Some-iff* **by** *auto*
have *ia-less-card*: $ia < CARD('a)$ **using** *ia-less-length* **by** (*metis diff-zero length-rev length-upt length-vec-to-iarray*)
have *ia-less-length'*: $ia < length\ ([0..{<}IArray.length\ (vec\text{-}to\text{-}iarray\ A)])$ **using** *ia-less-length*
unfolding *length-rev* **.**
have *a-less-card*: $a < CARD('a)$ **unfolding** *a-eq* **unfolding** *rev-nth*[*OF ia-less-length'*]
using *nth-upt*[*of 0* (*length* [*0..<IArray.length* (*vec-to-iarray A*)] − *Suc ia*) (*length* [*0..<IArray.length* (*vec-to-iarray A*)])]
by (*metis diff-less length-upt length-vec-to-iarray minus-nat.diff-0 plus-nat.add-0 zero-less-Suc zero-less-card-finite*)
have $(GREATEST\ b.\ A \$ b \neq 0) = from\text{-}nat\ a$
proof (*rule Greatest-equality*)
have $A \$ from\text{-}nat\ a = (vec\text{-}to\text{-}iarray\ A)\ !!\ a$ **by** (*rule vec-to-iarray-nth*[*symmetric*,*OF*

a-less-card])
also have ... ≠ *0* **using** *P-xs-ia* **unfolding** *a-eq*[*symmetric*] **.**
finally show *A* $ *from-nat a* ≠ *0* **.**
next
fix *y* **assume** *Ay*: *A* $ *y* ≠ *0*
show *y* ≤ *from-nat a*
proof (*rule ccontr*)
assume ¬ *y* ≤ *from-nat a* **hence** *y-greater-a*: *y* > *from-nat a* **by** *simp*
have *y-greater-a'*: *to-nat y* > *a* **using** *y-greater-a* **using** *to-nat-mono*[*of from-nat a y*] **using** *to-nat-from-nat-id* **by** (*metis a-less-card*)
have *a* = *?xs* ! *ia* **using** *a-eq* **.**
also have ... = [*0*..<*IArray.length* (*vec-to-iarray A*)] ! (*length* [*0*..<*IArray.length* (*vec-to-iarray A*)] − *Suc ia*) **by** (*rule rev-nth*[*OF ia-less-length'*])
also have ... = *0* + (*length* [*0*..<*IArray.length* (*vec-to-iarray A*)] − *Suc ia*) **apply** (*rule nth-upt*) **using** *ia-less-length'* **by** *fastforce*
also have ... = (*length* [*0*..<*IArray.length* (*vec-to-iarray A*)] − *Suc ia*) **by** *simp*
finally have *a* = (*length* [*0*..<*IArray.length* (*vec-to-iarray A*)] − *Suc ia*) **.**
hence *ia-eq*: *ia* = *length* [*0*..<*IArray.length* (*vec-to-iarray A*)] − (*Suc a*)
by (*metis Suc-diff-Suc Suc-eq-plus1-left diff-diff-cancel less-imp-le ia-less-length length-rev*)
define *ja* **where** *ja* = *length* [*0*..<*IArray.length* (*vec-to-iarray A*)] − *to-nat y* − *1*
have *ja-less-length*: *ja* < *length* [*0*..<*IArray.length* (*vec-to-iarray A*)] **unfolding** *ja-def*
using *ia-eq ia-less-length'* **by** (*simp add*: *algebra-simps*)
have *suc-i-le*: *IArray.length* (*vec-to-iarray A*)≥*Suc* (*to-nat y*) **unfolding** *vec-to-iarray-def* **using** *to-nat-less-card*[*of y*] **by** *auto*
have *?xs* ! *ja* = [*0*..<*IArray.length* (*vec-to-iarray A*)] ! (*length* [*0*..<*IArray.length* (*vec-to-iarray A*)] − *Suc ja*) **unfolding** *rev-nth*[*OF ja-less-length*] **..**
also have ... = *0* + (*length* [*0*..<*IArray.length* (*vec-to-iarray A*)] − *Suc ja*) **apply** (*rule nth-upt*, *auto simp del*: *IArray.length-def*) **unfolding** *ja-def*
by (*metis diff-Suc-less ia-less-length' length-upt less-nat-zero-code minus-nat.diff-0 neq0-conv*)
also have ... = (*length* [*0*..<*IArray.length* (*vec-to-iarray A*)] − *Suc ja*) **by** *simp*
also have ... = *to-nat y* **unfolding** *ja-def* **using** *suc-i-le* **by** *force*
finally have *xs-ja-eq-y*: *?xs* ! *ja* = *to-nat y* **.**
have *ja-less-ia*: *ja* < *ia* **unfolding** *ja-def ia-eq* **by** (*auto simp del*: *IArray.length-def*, *metis Suc-leI suc-i-le diff-less-mono2 le-imp-less-Suc less-le-trans y-greater-a'*)
hence *eq-0*: *vec-to-iarray A* !! (*?xs* ! *ja*) = *0* **using** *all-zero* **by** *simp*
hence *A* $ *y* = *0* **using** *vec-to-iarray-nth'*[*of A y*] **unfolding** *xs-ja-eq-y* **by** *simp*
thus *False* **using** *Ay* **by** *contradiction*
qed
qed
thus *?thesis* **unfolding** *greatest-not-zero-def a* **unfolding** *to-nat-eq*[*symmetric*] **unfolding** *to-nat-from-nat-id*[*OF a-less-card*] **by** *simp*

qed

20.2 Consistency and inconsistency

definition *consistent-iarrays A b* = (*let GJ*=*Gauss-Jordan-iarrays-PA A*;
rank-A = *length* [*x*←*IArray.list-of* (*snd GJ*) . ¬ *is-zero-iarray x*];
P-mult-b = *fst*(*GJ*) ∗*iv b*
in (*rank-A* ≥ (*if* (¬ *is-zero-iarray P-mult-b*)
then (*greatest-not-zero P-mult-b* + *1*) *else 0*)))

definition *inconsistent-iarrays A b* = (¬ *consistent-iarrays A b*)

lemma *matrix-to-iarray-consistent*[*code*]: *consistent A b* = *consistent-iarrays* (*matrix-to-iarray A*) (*vec-to-iarray b*)
unfolding *consistent-eq-rank-ge-code*
unfolding *consistent-iarrays-def Let-def*
unfolding *Gauss-Jordan-PA-eq*
unfolding *rank-Gauss-Jordan-code*[*symmetric*, *unfolded Let-def*]
unfolding *snd-Gauss-Jordan-iarrays-PA-eq*
unfolding *rank-iarrays-code*[*symmetric*]
unfolding *matrix-to-iarray-rank*
unfolding *matrix-to-iarray-fst-Gauss-Jordan-PA*[*symmetric*]
unfolding *vec-to-iarray-matrix-matrix-mult*[*symmetric*]
unfolding *not-is-zero-iarray-eq-iff*
using *vec-to-iarray-greatest-not-zero*[*unfolded not-is-zero-iarray-eq-iff*]
by *force*

lemma *matrix-to-iarray-inconsistent*[*code*]: *inconsistent A b* = *inconsistent-iarrays* (*matrix-to-iarray A*) (*vec-to-iarray b*)
unfolding *inconsistent-def inconsistent-iarrays-def*
unfolding *matrix-to-iarray-consistent* **..**

definition *solve-consistent-rref-iarrays A b*
= *IArray.of-fun* (λ*j*. *if* (*IArray.exists* (λ*i*. *A* !! *i* !! *j* = *1* ∧ *j*=*least-non-zero-position-of-vector* (*row-iarray i A*)) (*IArray*[*0*..<*nrows-iarray A*]))
then b !! (*least-non-zero-position-of-vector* (*column-iarray j A*)) *else 0*) (*ncols-iarray A*)

lemma *exists-solve-consistent-rref*:
fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *rref*: *reduced-row-echelon-form A*
shows (∃ *i*. *A* $ *i* $ *j* = *1* ∧ *j* = (*LEAST n*. *A* $ *i* $ *n* ≠ *0*))
= (*IArray.exists* (λ*i*. (*matrix-to-iarray A*) !! *i* !! (*to-nat j*) = *1*
∧ (*to-nat j*)=*least-non-zero-position-of-vector* (*row-iarray i* (*matrix-to-iarray A*)))
(*IArray*[*0*..<*nrows-iarray* (*matrix-to-iarray A*)]))
proof (*rule*)

assume $\exists i.\ A\ \$\ i\ \$\ j = 1 \wedge j = (LEAST\ n.\ A\ \$\ i\ \$\ n \neq 0)$
from *this* **obtain** *i* **where** *Aij*: $A\ \$\ i\ \$\ j = 1$ **and** *j-eq*: $j = (LEAST\ n.\ A\ \$\ i\ \$\ n \neq 0)$ **by** *blast*
show *IArray.exists* $(\lambda i.\ matrix\text{-}to\text{-}iarray\ A\ !!\ i\ !!\ to\text{-}nat\ j = 1 \wedge to\text{-}nat\ j = least\text{-}non\text{-}zero\text{-}position\text{-}of\text{-}vector\ (row\text{-}iarray\ i\ (matrix\text{-}to\text{-}iarray\ A)))$
(*IArray* $[0..<nrows\text{-}iarray\ (matrix\text{-}to\text{-}iarray\ A)]$)
unfolding *IArray.exists-def find-Some-iff*
apply (*rule bexI*[*of* - *to-nat i*])+
proof (*auto*, *unfold IArray.sub-def*[*symmetric*])
show *to-nat i* < *nrows-iarray* (*matrix-to-iarray A*) **unfolding** *matrix-to-iarray-nrows*[*symmetric*] *nrows-def* **using** *to-nat-less-card* **by** *fast*
have *to-nat j* = *to-nat* (*LEAST n. A* $ *i* $ *n* ≠ *0*) **unfolding** *j-eq* **by** *simp*
also have ... = *to-nat* (*LEAST n. A* $ *i* $ *n* ≠ *0* ∧ *0*≤*n*) **by** (*metis least-mod-type*)
also have ...= *least-non-zero-position-of-vector-from-index* (*vec-to-iarray* (*row i A*)) (*to-nat* (*0*::*'cols*))
proof (*rule vec-to-iarray-least-non-zero-position-of-vector-from-index''*[*symmetric*, *of 0*::*'cols i A*])
show ¬ *vector-all-zero-from-index* (*to-nat* (*0*::*'cols*), *vec-to-iarray* (*row i A*))
unfolding *vector-all-zero-from-index-eq*[*symmetric*, *of 0*::*'cols row i A*] **unfolding** *row-def vec-nth-inverse* **using** *Aij least-mod-type*[*of j*] **by** *fastforce*
qed
also have ... = *least-non-zero-position-of-vector* (*row-iarray* (*to-nat i*) (*matrix-to-iarray A*)) **unfolding** *vec-to-iarray-row least-non-zero-position-of-vector-def*
unfolding *to-nat-0* **..**
finally show *to-nat j* = *least-non-zero-position-of-vector* (*row-iarray* (*to-nat i*) (*matrix-to-iarray A*)) **.**
show *matrix-to-iarray A* !! *mod-type-class.to-nat i* !! *mod-type-class.to-nat j* = *1* **unfolding** *matrix-to-iarray-nth* **using** *Aij* **.**
qed
next
assume *ex-eq*: *IArray.exists* $(\lambda i.\ matrix\text{-}to\text{-}iarray\ A\ !!\ i\ !!\ to\text{-}nat\ j = 1 \wedge to\text{-}nat\ j = least\text{-}non\text{-}zero\text{-}position\text{-}of\text{-}vector\ (row\text{-}iarray\ i\ (matrix\text{-}to\text{-}iarray\ A)))$
(*IArray* $[0..<nrows\text{-}iarray\ (matrix\text{-}to\text{-}iarray\ A)]$)
have $\exists y.\ List.find\ (\lambda i.\ matrix\text{-}to\text{-}iarray\ A\ !!\ i\ !!\ to\text{-}nat\ j = 1 \wedge to\text{-}nat\ j = least\text{-}non\text{-}zero\text{-}position\text{-}of\text{-}vector\ (row\text{-}iarray\ i\ (matrix\text{-}to\text{-}iarray\ A)))$
$[0..<nrows\text{-}iarray\ (matrix\text{-}to\text{-}iarray\ A)] = Some\ y$
proof (*rule ccontr*, *simp del*: *IArray.length-def IArray.sub-def*, *unfold find-None-iff*)
assume ¬ $(\exists x.\ x \in set\ [0..<nrows\text{-}iarray\ (matrix\text{-}to\text{-}iarray\ A)] \wedge$
$matrix\text{-}to\text{-}iarray\ A\ !!\ x\ !!\ mod\text{-}type\text{-}class.to\text{-}nat\ j = 1 \wedge mod\text{-}type\text{-}class.to\text{-}nat\ j = least\text{-}non\text{-}zero\text{-}position\text{-}of\text{-}vector\ (row\text{-}iarray\ x\ (matrix\text{-}to\text{-}iarray\ A)))$
thus *False* **using** *ex-eq* **unfolding** *IArray.exists-def* **by** *auto*
qed
from *this* **obtain** *y* **where** *y*: $List.find\ (\lambda i.\ matrix\text{-}to\text{-}iarray\ A\ !!\ i\ !!\ to\text{-}nat\ j = 1 \wedge to\text{-}nat\ j = least\text{-}non\text{-}zero\text{-}position\text{-}of\text{-}vector\ (row\text{-}iarray\ i\ (matrix\text{-}to\text{-}iarray\ A)))$
$[0..<nrows\text{-}iarray\ (matrix\text{-}to\text{-}iarray\ A)] = Some\ y$ **by** *blast*

from *this* **obtain** *i* **where** *i-less-length*: *i*<*length* [*0*..<*nrows-iarray* (*matrix-to-iarray A*)] **and**
Aij-1: *matrix-to-iarray A* !! ([*0*..<*nrows-iarray* (*matrix-to-iarray A*)] ! *i*) !! *to-nat j* = *1*
and *j-eq*: *to-nat j* = *least-non-zero-position-of-vector* (*row-iarray* ([*0*..<*nrows-iarray* (*matrix-to-iarray A*)] ! *i*) (*matrix-to-iarray A*)) **and**
y-eq: *y* = [*0*..<*nrows-iarray* (*matrix-to-iarray A*)] ! *i*
and *least*: (∀ *ja*<*i*. ¬ (*matrix-to-iarray A* !! ([*0*..<*nrows-iarray* (*matrix-to-iarray A*)] ! *ja*) !! *to-nat j* = *1* ∧
to-nat j = *least-non-zero-position-of-vector* (*row-iarray* ([*0*..<*nrows-iarray* (*matrix-to-iarray A*)] ! *ja*) (*matrix-to-iarray A*))))
unfolding *find-Some-iff* **by** *blast*
show ∃ *i*. *A* $ *i* $ *j* = *1* ∧ *j* = (*LEAST n*. *A* $ *i* $ *n* ≠ *0*)
proof (*rule exI*[*of* - *from-nat i*], *rule conjI*)
have *i-rw*: [*0*..<*nrows-iarray* (*matrix-to-iarray A*)] ! *i* = *i* **using** *nth-upt*[*of 0 i nrows-iarray* (*matrix-to-iarray A*)] **using** *i-less-length* **by** *auto*
have *i-less-card*: *i* < *CARD* ('*rows*) **using** *i-less-length* **unfolding** *nrows-iarray-def matrix-to-iarray-def* **by** *auto*
show *A-ij*: *A* $ *from-nat i* $ *j* = *1*
using *Aij-1* **unfolding** *i-rw* **using** *matrix-to-iarray-nth*[*of A from-nat i j*] **unfolding** *to-nat-from-nat-id*[*OF i-less-card*] **by** *simp*
have *to-nat j* = *least-non-zero-position-of-vector* (*row-iarray* ([*0*..<*nrows-iarray* (*matrix-to-iarray A*)] ! *i*) (*matrix-to-iarray A*)) **using** *j-eq* **.**
also have ... = *least-non-zero-position-of-vector-from-index* (*row-iarray i* (*matrix-to-iarray A*)) *0*
unfolding *least-non-zero-position-of-vector-def i-rw* **..**
also have ... = *least-non-zero-position-of-vector-from-index* (*vec-to-iarray* (*row* (*from-nat i*) *A*)) (*to-nat* (*0*::'*cols*)) **unfolding** *vec-to-iarray-row*
unfolding *to-nat-from-nat-id*[*OF i-less-card*] **unfolding** *to-nat-0* **..**
also have ... = *to-nat* (*LEAST n*. *A* $ (*from-nat i*) $ *n* ≠ *0* ∧ *0* ≤ *n*)
proof (*rule vec-to-iarray-least-non-zero-position-of-vector-from-index''*)
show ¬ *vector-all-zero-from-index* (*to-nat* (*0*::'*cols*)), *vec-to-iarray* (*row* (*from-nat i*) *A*))
unfolding *vector-all-zero-from-index-eq*[*symmetric*] **using** *A-ij* **by** (*metis iarray-to-vec-vec-to-iarray least-mod-type vec-matrix vec-to-iarray-row' zero-neq-one*)
qed
also have ... = *to-nat* (*LEAST n*. *A* $ (*from-nat i*) $ *n* ≠ *0*) **using** *least-mod-type* **by** *metis*
finally show *j* = (*LEAST n*. *A* $ *from-nat i* $ *n* ≠ *0*) **unfolding** *to-nat-eq* **.**
qed
qed

lemma *to-nat-the-solve-consistent-rref*:
fixes *A*::'*a*::{*field*}^'*cols*::{*mod-type*}^'*rows*::{*mod-type*}
assumes *rref*: *reduced-row-echelon-form A*
and *exists*: (∃ *i*. *A* $ *i* $ *j* = *1* ∧ *j* = (*LEAST n*. *A* $ *i* $ *n* ≠ *0*))
shows *to-nat* (*THE i*. *A* $ *i* $ *j* = *1*) = *least-non-zero-position-of-vector* (*column-iarray* (*to-nat j*) (*matrix-to-iarray A*))

proof −
obtain *i* **where** *Aij*: *A* $ *i* $ *j* = *1* **and** *j*:*j* = (*LEAST n*. *A* $ *i* $ *n* ≠ *0*) **using** *exists* **by** *blast*
have *least-non-zero-position-of-vector* (*column-iarray* (*to-nat j*) (*matrix-to-iarray A*)) =
least-non-zero-position-of-vector (*vec-to-iarray* (*column j A*)) **unfolding** *vec-to-iarray-column*
..
also have ... = *least-non-zero-position-of-vector-from-index* (*vec-to-iarray* (*column j A*)) (*to-nat* (*0*::*'rows*)) **unfolding** *least-non-zero-position-of-vector-def to-nat-0* **..**
also have ... = *to-nat* (*LEAST n*. *A* $ *n* $ *j* ≠ *0* ∧ *0* ≤ *n*)
proof (*rule vec-to-iarray-least-non-zero-position-of-vector-from-index'*)
show ¬ *vector-all-zero-from-index* (*to-nat* (*0*::*'rows*), *vec-to-iarray* (*column j A*))
unfolding *vector-all-zero-from-index-eq*[*symmetric*] *column-def* **using** *Aij least-mod-type*[*of i*] **by** *fastforce*
qed
also have ... = *to-nat* (*LEAST n*. *A* $ *n* $ *j* ≠ *0*) **using** *least-mod-type* **by** *metis*
finally have *least-eq*: *least-non-zero-position-of-vector* (*column-iarray* (*to-nat j*) (*matrix-to-iarray A*)) = *to-nat* (*LEAST n*. *A* $ *n* $ *j* ≠ *0*) **.**
have *i-eq-least*: *i*=(*LEAST n*. *A* $ *n* $ *j* ≠ *0*)
proof (*rule Least-equality*[*symmetric*])
show *A* $ *i* $ *j* ≠ *0* **by** (*metis Aij zero-neq-one*)
show ⋀*y*. *A* $ *y* $ *j* ≠ *0* ⟹ *i* ≤ *y* **by** (*metis* (*mono-tags*) *Aij is-zero-row-def' j order-refl rref rref-condition4 zero-neq-one*)
qed
have *the-eq-least-pos*: (*THE i*. *A* $ *i* $ *j* = *1*) = *from-nat* (*least-non-zero-position-of-vector* (*column-iarray* (*to-nat j*) (*matrix-to-iarray A*)))
proof (*rule the-equality*)
show *A* $ *from-nat* (*least-non-zero-position-of-vector* (*column-iarray* (*to-nat j*) (*matrix-to-iarray A*))) $ *j* = *1*
unfolding *least-eq from-nat-to-nat-id i-eq-least*[*symmetric*] **using** *Aij* **.**
fix *a* **assume** *a*: *A* $ *a* $ *j* = *1*
show *a* = *from-nat* (*least-non-zero-position-of-vector* (*column-iarray* (*to-nat j*) (*matrix-to-iarray A*)))
unfolding *least-eq from-nat-to-nat-id*
by (*metis Aij a i-eq-least is-zero-row-def' j rref rref-condition4-explicit zero-neq-one*)
qed
have *to-nat* (*THE i*. *A* $ *i* $ *j* = *1*) = *to-nat* (*from-nat* (*least-non-zero-position-of-vector* (*column-iarray* (*to-nat j*) (*matrix-to-iarray A*)))::*'rows*) **using** *the-eq-least-pos* **by** *auto*
also have ... = (*least-non-zero-position-of-vector* (*column-iarray* (*to-nat j*) (*matrix-to-iarray A*))) **by** (*rule to-nat-from-nat-id*, *unfold least-eq*, *simp add*: *to-nat-less-card*)
also have ... = *to-nat* (*LEAST n*. *A* $ *n* $ *j* ≠ *0*) **unfolding** *least-eq from-nat-to-nat-id*
..
finally have (*THE i*. *A* $ *i* $ *j* = *1*) = (*LEAST n*. *A* $ *n* $ *j* ≠ *0*) **unfolding** *to-nat-eq* **.**
thus *?thesis* **unfolding** *least-eq from-nat-to-nat-id* **unfolding** *to-nat-eq* **.**
qed

lemma *iarray-exhaust2*:
(*xs* = *ys*) = (*IArray.list-of xs* = *IArray.list-of ys*)
by (*metis iarray.exhaust list-of.simps*)

lemma *vec-to-iarray-solve-consistent-rref*:
fixes *A*::'*a*::{*field*}^'*cols*::{*mod-type*}^'*rows*::{*mod-type*}
assumes *rref*: *reduced-row-echelon-form A*
shows *vec-to-iarray* (*solve-consistent-rref A b*) = *solve-consistent-rref-iarrays* (*matrix-to-iarray A*) (*vec-to-iarray b*)
proof(*unfold iarray-exhaust2 list-eq-iff-nth-eq IArray.length-def*[*symmetric*] *IArray.sub-def*[*symmetric*], *rule conjI*)
show *IArray.length* (*vec-to-iarray* (*solve-consistent-rref A b*)) = *IArray.length* (*solve-consistent-rref-iarrays* (*matrix-to-iarray A*) (*vec-to-iarray b*))
unfolding *solve-consistent-rref-def solve-consistent-rref-iarrays-def*
unfolding *ncols-iarray-def matrix-to-iarray-def* **by** (*simp add*: *vec-to-iarray-def*)
show ∀ *i*<*IArray.length* (*vec-to-iarray* (*solve-consistent-rref A b*)). *vec-to-iarray* (*solve-consistent-rref A b*) !! *i* = *solve-consistent-rref-iarrays* (*matrix-to-iarray A*) (*vec-to-iarray b*) !! *i*
proof (*clarify*)
fix *i* **assume** *i*: *i* < *IArray.length* (*vec-to-iarray* (*solve-consistent-rref A b*))
hence *i-less-card*: *i*<*CARD*('*cols*) **unfolding** *vec-to-iarray-def* **by** *auto*
hence *i-less-ncols*: *i*<(*ncols-iarray* (*matrix-to-iarray A*)) **unfolding** *ncols-eq-card-columns*
.
show *vec-to-iarray* (*solve-consistent-rref A b*) !! *i* = *solve-consistent-rref-iarrays* (*matrix-to-iarray A*) (*vec-to-iarray b*) !! *i*
unfolding *vec-to-iarray-nth*[*OF i-less-card*]
unfolding *solve-consistent-rref-def*
unfolding *vec-lambda-beta*
unfolding *solve-consistent-rref-iarrays-def*
unfolding *of-fun-nth*[*OF i-less-ncols*]
unfolding *exists-solve-consistent-rref*[*OF rref*, *of from-nat i*, *symmetric*, *unfolded to-nat-from-nat-id*[*OF i-less-card*]]
using *to-nat-the-solve-consistent-rref*[*OF rref*, *of from-nat i*, *symmetric*, *unfolded to-nat-from-nat-id*[*OF i-less-card*]]
using *vec-to-iarray-nth'* **by** *metis*
qed
qed

20.3 Independence and dependence

definition *independent-and-consistent-iarrays A b* =
(*let GJ* = *Gauss-Jordan-iarrays-PA A*;
rank-A = *length* [*x*←*IArray.list-of* (*snd GJ*) . ¬ *is-zero-iarray x*];
P-mult-b = *fst GJ* ∗*iv b*;
consistent-A = ((*if* ¬ *is-zero-iarray P-mult-b then greatest-not-zero P-mult-b* + *1 else 0*) ≤ *rank-A*);
dim-solution-set = *ncols-iarray A* − *rank-A*

in consistent-A $\wedge$ *dim-solution-set* $= 0$)

definition *dependent-and-consistent-iarrays A b* =
(*let GJ* = *Gauss-Jordan-iarrays-PA A*;
rank-A = *length* [*x*←*IArray.list-of* (*snd GJ*) . $\neg$ *is-zero-iarray x*];
P-mult-b = *fst GJ* $*_{iv}$ *b*;
consistent-A = ((*if* $\neg$ *is-zero-iarray P-mult-b then greatest-not-zero P-mult-b* + *1 else 0*) $\leq$ *rank-A*);
dim-solution-set = *ncols-iarray A* − *rank-A*
in consistent-A $\wedge$ *dim-solution-set* > 0)

lemma *matrix-to-iarray-independent-and-consistent*[*code*]:
shows *independent-and-consistent A b* = *independent-and-consistent-iarrays* (*matrix-to-iarray A*) (*vec-to-iarray b*)
unfolding *independent-and-consistent-def*
unfolding *independent-and-consistent-iarrays-def*
unfolding *dim-solution-set-homogeneous-eq-dim-null-space*
unfolding *matrix-to-iarray-consistent*
unfolding *consistent-iarrays-def*
unfolding *dim-null-space-iarray*
unfolding *rank-iarrays-code*
unfolding *snd-Gauss-Jordan-iarrays-PA-eq*[*symmetric*]
unfolding *Let-def* **..**

lemma *matrix-to-iarray-dependent-and-consistent*[*code*]:
shows *dependent-and-consistent A b* = *dependent-and-consistent-iarrays* (*matrix-to-iarray A*) (*vec-to-iarray b*)
unfolding *dependent-and-consistent-def*
unfolding *dependent-and-consistent-iarrays-def*
unfolding *dim-solution-set-homogeneous-eq-dim-null-space*
unfolding *matrix-to-iarray-consistent*
unfolding *consistent-iarrays-def*
unfolding *dim-null-space-iarray*
unfolding *rank-iarrays-code*
unfolding *snd-Gauss-Jordan-iarrays-PA-eq*[*symmetric*]
unfolding *Let-def* **..**

20.4 Solve a system of equations over nested IArrays

definition *solve-system-iarrays A b* = (*let A′* = *Gauss-Jordan-iarrays-PA A in* (*snd A′*, *fst A′* $*_{iv}$ *b*))

lemma *matrix-to-iarray-fst-solve-system*: *matrix-to-iarray* (*fst* (*solve-system A b*)) = *fst* (*solve-system-iarrays* (*matrix-to-iarray A*) (*vec-to-iarray b*))
unfolding *solve-system-def solve-system-iarrays-def Let-def fst-conv*
by (*metis matrix-to-iarray-snd-Gauss-Jordan-PA*)

lemma *vec-to-iarray-snd-solve-system*: *vec-to-iarray* (*snd* (*solve-system A b*)) =

snd (*solve-system-iarrays* (*matrix-to-iarray A*) (*vec-to-iarray b*))
unfolding *solve-system-def solve-system-iarrays-def Let-def snd-conv*
by (*metis matrix-to-iarray-fst-Gauss-Jordan-PA vec-to-iarray-matrix-matrix-mult*)

definition *solve-iarrays A b* = (*let GJ-P*=*Gauss-Jordan-iarrays-PA A*;
P-mult-b = *fst GJ-P* ∗*iv b*;
rank-A = *length* [*x*←*IArray.list-of* (*snd GJ-P*) . ¬ *is-zero-iarray x*];
consistent-Ab = (*if* ¬ *is-zero-iarray P-mult-b then greatest-not-zero P-mult-b* + *1 else 0*) ≤ *rank-A*;
GJ-transpose = *Gauss-Jordan-iarrays-PA* (*transpose-iarray A*);
basis = *set* (*map* (λ*i*. *row-iarray i* (*fst GJ-transpose*)) [*rank-A*..<*ncols-iarray A*])
in (*if consistent-Ab then Some* (*solve-consistent-rref-iarrays* (*snd GJ-P*) *P-mult-b*,*basis*) *else None*))

definition *pair-vec-vecset A* = (*if Option.is-none A then None else Some* (*vec-to-iarray* (*fst* (*the A*)), *vec-to-iarray*‘ (*snd* (*the A*))))

lemma *pair-vec-vecset-solve*[*code-unfold*]:
shows *pair-vec-vecset* (*solve A b*) = *solve-iarrays* (*matrix-to-iarray A*) (*vec-to-iarray b*)
unfolding *pair-vec-vecset-def*
proof (*auto*)
assume *none-solve-Ab*: *Option.is-none* (*solve A b*)
show *None* = *solve-iarrays* (*matrix-to-iarray A*) (*vec-to-iarray b*)
proof −
define *GJ-P* **where** *GJ-P* = *Gauss-Jordan-iarrays-PA* (*matrix-to-iarray A*)
define *P-mult-b* **where** *P-mult-b* = *fst GJ-P* ∗*iv vec-to-iarray b*
define *rank-A* **where** *rank-A* = *length* [*x*←*IArray.list-of* (*snd GJ-P*). ¬ *is-zero-iarray x*]
have ¬ *consistent A b* **using** *none-solve-Ab* **unfolding** *solve-def* **unfolding** *Option.is-none-def* **by** *auto*
hence ¬ *consistent-iarrays* (*matrix-to-iarray A*) (*vec-to-iarray b*) **using** *matrix-to-iarray-consistent* **by** *auto*
hence ¬ (*if* ¬ *is-zero-iarray P-mult-b then greatest-not-zero P-mult-b* + *1 else 0*) ≤ *rank-A*
unfolding *GJ-P-def P-mult-b-def rank-A-def*
using *consistent-iarrays-def* **unfolding** *Let-def* **by** *fast*
thus *?thesis* **unfolding** *solve-iarrays-def Let-def* **unfolding** *GJ-P-def P-mult-b-def rank-A-def* **by** *presburger*
qed
next
assume *not-none*: ¬ *Option.is-none* (*solve A b*)
show *Some* (*vec-to-iarray* (*fst* (*the* (*solve A b*))), *vec-to-iarray* ‘ *snd* (*the* (*solve A b*))) = *solve-iarrays* (*matrix-to-iarray A*) (*vec-to-iarray b*)
proof −
define *GJ-P* **where** *GJ-P* = *Gauss-Jordan-iarrays-PA* (*matrix-to-iarray A*)

define *P-mult-b* **where** *P-mult-b* = *fst GJ-P* ∗*iv vec-to-iarray b*
define *rank-A* **where** *rank-A* = *length* [*x*←*IArray.list-of* (*snd GJ-P*) . ¬ *is-zero-iarray x*]
define *GJ-transpose* **where** *GJ-transpose* = *Gauss-Jordan-iarrays-PA* (*transpose-iarray* (*matrix-to-iarray A*))
define *basis* **where** *basis* = *set* (*map* (λ*i*. *row-iarray i* (*fst GJ-transpose*)) [*rank-A*..<*ncols-iarray* (*matrix-to-iarray A*)])
define *P-mult-b* **where** *P-mult-b* = *fst GJ-P* ∗*iv vec-to-iarray b*
have *consistent-Ab*: *consistent A b* **using** *not-none* **unfolding** *solve-def* **unfolding** *Option.is-none-def* **by** *metis*
hence *consistent-iarrays* (*matrix-to-iarray A*) (*vec-to-iarray b*) **using** *matrix-to-iarray-consistent* **by** *auto*
hence (*if* ¬ *is-zero-iarray P-mult-b then greatest-not-zero P-mult-b* + *1 else 0*) ≤ *rank-A*
unfolding *GJ-P-def P-mult-b-def rank-A-def*
using *consistent-iarrays-def* **unfolding** *Let-def* **by** *fast*
hence *solve-iarrays-rw*: *solve-iarrays* (*matrix-to-iarray A*) (*vec-to-iarray b*) = *Some* (*solve-consistent-rref-iarrays* (*snd GJ-P*) *P-mult-b*, *basis*)
unfolding *solve-iarrays-def Let-def P-mult-b-def GJ-P-def rank-A-def basis-def GJ-transpose-def* **by** *auto*
have *snd-rw*: *vec-to-iarray* ‘ *basis-null-space A* = *basis* **unfolding** *basis-def GJ-transpose-def rank-A-def GJ-P-def*
unfolding *vec-to-iarray-basis-null-space* **unfolding** *basis-null-space-iarrays-def Let-def*
unfolding *snd-Gauss-Jordan-iarrays-PA-eq*
unfolding *rank-iarrays-code*[*symmetric*]
unfolding *matrix-to-iarray-transpose*[*symmetric*]
unfolding *matrix-to-iarray-rank*[*symmetric*]
unfolding *rank-transpose*[*symmetric*, *of A*] **..**
have *fst-rw*: *vec-to-iarray* (*solve-consistent-rref* (*fst* (*solve-system A b*)) (*snd* (*solve-system A b*))) = *solve-consistent-rref-iarrays* (*snd GJ-P*) *P-mult-b*
using *vec-to-iarray-solve-consistent-rref*[*OF rref-Gauss-Jordan*, *of A fst* (*Gauss-Jordan-PA A*) ∗*v b*]
unfolding *solve-system-def Let-def fst-conv*
unfolding *Gauss-Jordan-PA-eq snd-conv*
unfolding *GJ-P-def P-mult-b-def*
unfolding *vec-to-iarray-matrix-matrix-mult*
unfolding *matrix-to-iarray-fst-Gauss-Jordan-PA*[*symmetric*]
unfolding *matrix-to-iarray-snd-Gauss-Jordan-PA*[*symmetric*]
unfolding *Gauss-Jordan-PA-eq* **.**
show *?thesis* **unfolding** *solve-iarrays-rw*
unfolding *solve-def if-P*[*OF consistent-Ab*] *option.sel fst-conv snd-conv*
unfolding *fst-rw snd-rw* **..**
qed
qed

end

21 Computing determinants of matrices using the Gauss Jordan algorithm over nested IArrays

theory *Determinants-IArrays*
imports
Determinants2
Gauss-Jordan-IArrays
begin

21.1 Definitions

definition *Gauss-Jordan-in-ij-det-P-iarrays A i j* = (*let n* = *least-non-zero-position-of-vector-from-index* (*column-iarray j A*) *i*
in (*if i* = *n then 1 / A !! i !! j else* − *1 / A !! n !! j*, *Gauss-Jordan-in-ij-iarrays A i j*))

definition *Gauss-Jordan-column-k-det-P-iarrays A' k* = (*let det-P* = *fst A'*; *i* = *fst* (*snd A'*); *A* = *snd* (*snd A'*)
in if vector-all-zero-from-index (*i*, *column-iarray k A*) ∨ *i* = *nrows-iarray A then* (*det-P*, *i*, *A*)
else let gauss = *Gauss-Jordan-in-ij-det-P-iarrays A i k in* (*fst gauss* ∗ *det-P*, *i* + *1*, *snd gauss*))

definition *Gauss-Jordan-upt-k-det-P-iarrays A k* = (*let foldl* = *foldl Gauss-Jordan-column-k-det-P-iarrays* (*1*, *0*, *A*) [*0..<Suc k*] *in* (*fst foldl*, *snd* (*snd foldl*)))
definition *Gauss-Jordan-det-P-iarrays A* = *Gauss-Jordan-upt-k-det-P-iarrays A* (*ncols-iarray A* − *1*)

21.2 Proofs

A more efficient equation for *Gauss-Jordan-in-ij-det-P-iarrays A i j*.

lemma *Gauss-Jordan-in-ij-det-P-iarrays-code*[*code*]: *Gauss-Jordan-in-ij-det-P-iarrays A i j*
= (*let n* = *least-non-zero-position-of-vector-from-index* (*column-iarray j A*) *i*;
interchange-A = *interchange-rows-iarray A i n*;
A' = *mult-row-iarray interchange-A i* (*1 / interchange-A !! i !! j*)
in (*if i* = *n then 1 / A !! i !! j else* − *1 / A !! n !! j*, *IArray.of-fun* (λ*s. if s* = *i then A' !! s else row-add-iarray A' s i* (− *interchange-A !! s !! j*) *!! s*) (*nrows-iarray A*)))
unfolding *Gauss-Jordan-in-ij-det-P-iarrays-def Gauss-Jordan-in-ij-iarrays-def Let-def*
..

lemma *matrix-to-iarray-fst-Gauss-Jordan-in-ij-det-P*:
assumes *ex-n*: ∃ *n*. *A $ n $ j* ≠ *0* ∧ *i* ≤ *n*
shows *fst* (*Gauss-Jordan-in-ij-det-P A i j*) = *fst* (*Gauss-Jordan-in-ij-det-P-iarrays* (*matrix-to-iarray A*) (*to-nat i*) (*to-nat j*))
proof −

have *least-n-eq*: *least-non-zero-position-of-vector-from-index* (*vec-to-iarray* (*column*
j A)) (*to-nat i*) = *to-nat* (*LEAST n. A $ n $ j* ≠ *0* ∧ *i* ≤ *n*)
by (*rule vec-to-iarray-least-non-zero-position-of-vector-from-index'*[*unfolded matrix-vector-all-zero-from-index*[
metis ex-n)
show *?thesis*
unfolding *Gauss-Jordan-in-ij-det-P-def Gauss-Jordan-in-ij-det-P-iarrays-def Let-def*
fst-conv
unfolding *least-n-eq*[*unfolded vec-to-iarray-column*] **unfolding** *matrix-to-iarray-nth*
by *auto*
qed

corollary *matrix-to-iarray-fst-Gauss-Jordan-in-ij-det-P'*:
assumes ¬ (*vector-all-zero-from-index* (*to-nat i*, *vec-to-iarray* (*column j A*)))
shows *fst* (*Gauss-Jordan-in-ij-det-P A i j*) = *fst* (*Gauss-Jordan-in-ij-det-P-iarrays*
(*matrix-to-iarray A*) (*to-nat i*) (*to-nat j*))
using *matrix-to-iarray-fst-Gauss-Jordan-in-ij-det-P assms* **unfolding** *matrix-vector-all-zero-from-index*[*symm*
by *auto*

lemma *matrix-to-iarray-snd-Gauss-Jordan-in-ij-det-P*:
assumes *ex-n*: ∃ *n. A $ n $ j* ≠ *0* ∧ *i* ≤ *n*
shows *matrix-to-iarray* (*snd* (*Gauss-Jordan-in-ij-det-P A i j*)) = *snd* (*Gauss-Jordan-in-ij-det-P-iarrays*
(*matrix-to-iarray A*) (*to-nat i*) (*to-nat j*))
unfolding *Gauss-Jordan-in-ij-det-P-def Gauss-Jordan-in-ij-det-P-iarrays-def Let-def*
snd-conv
by (*rule matrix-to-iarray-Gauss-Jordan-in-ij*, *simp add*: *matrix-vector-all-zero-from-index*[*symmetric*],
metis ex-n)

lemma *matrix-to-iarray-fst-Gauss-Jordan-column-k-det-P*:
assumes *i*: *i*≤*nrows A* **and** *k*: *k*<*ncols A*
shows *fst* (*Gauss-Jordan-column-k-det-P* (*n*, *i*, *A*) *k*) = *fst* (*Gauss-Jordan-column-k-det-P-iarrays*
(*n*, *i*, *matrix-to-iarray A*) *k*)
proof (*cases i*<*nrows A*)
case *True*
show *?thesis*
unfolding *Gauss-Jordan-column-k-det-P-def Gauss-Jordan-column-k-det-P-iarrays-def*
Let-def snd-conv fst-conv
unfolding *matrix-to-iarray-nrows matrix-vector-all-zero-from-index*
using *matrix-to-iarray-fst-Gauss-Jordan-in-ij-det-P'*[*of from-nat i from-nat k A*]
unfolding *vec-to-iarray-column*
unfolding *to-nat-from-nat-id*[*OF True*[*unfolded nrows-def*]]
unfolding *to-nat-from-nat-id*[*OF k*[*unfolded ncols-def*]]
by *auto*
next
case *False*
hence *i*=*nrows A* **using** *i* **by** *simp*
thus *?thesis*
unfolding *Gauss-Jordan-column-k-det-P-def Gauss-Jordan-column-k-det-P-iarrays-def*
Let-def snd-conv fst-conv **unfolding** *matrix-to-iarray-nrows* **by** *force*

qed

lemma *matrix-to-iarray-fst-snd-Gauss-Jordan-column-k-det-P*:
assumes *i*: *i*≤*nrows A* **and** *k*: *k*<*ncols A*
shows *fst* (*snd* (*Gauss-Jordan-column-k-det-P* (*n*, *i*, *A*) *k*)) = *fst* (*snd* (*Gauss-Jordan-column-k-det-P-iarrays* (*n*, *i*, *matrix-to-iarray A*) *k*))
proof (*cases i*<*nrows A*)
case *True*
show *?thesis*
unfolding *Gauss-Jordan-column-k-det-P-def Gauss-Jordan-column-k-det-P-iarrays-def Let-def snd-conv fst-conv*
unfolding *matrix-to-iarray-nrows matrix-vector-all-zero-from-index*
unfolding *vec-to-iarray-column*
unfolding *to-nat-from-nat-id*[*OF True*[*unfolded nrows-def*]]
unfolding *to-nat-from-nat-id*[*OF k*[*unfolded ncols-def*]] **by** *auto*
next
case *False*
thus *?thesis*
using *assms*
unfolding *Gauss-Jordan-column-k-det-P-def Gauss-Jordan-column-k-det-P-iarrays-def Let-def snd-conv fst-conv*
unfolding *matrix-to-iarray-nrows matrix-vector-all-zero-from-index* **by** *auto*
qed

lemma *matrix-to-iarray-snd-snd-Gauss-Jordan-column-k-det-P*:
assumes *i*: *i*≤*nrows A* **and** *k*: *k*<*ncols A*
shows *matrix-to-iarray* (*snd* (*snd* (*Gauss-Jordan-column-k-det-P* (*n*, *i*, *A*) *k*))) = *snd* (*snd* (*Gauss-Jordan-column-k-det-P-iarrays* (*n*, *i*, *matrix-to-iarray A*) *k*))
proof (*cases i*<*nrows A*)
case *True*
show *?thesis*
unfolding *Gauss-Jordan-column-k-det-P-def Gauss-Jordan-column-k-det-P-iarrays-def Let-def fst-conv snd-conv*
unfolding *matrix-to-iarray-nrows matrix-vector-all-zero-from-index*
using *matrix-to-iarray-snd-Gauss-Jordan-in-ij-det-P*[*of A from-nat k from-nat i*]
using *matrix-vector-all-zero-from-index*[*of from-nat i A from-nat k*]
unfolding *vec-to-iarray-column*
unfolding *to-nat-from-nat-id*[*OF True*[*unfolded nrows-def*]]
unfolding *to-nat-from-nat-id*[*OF k*[*unfolded ncols-def*]]
by *auto*
next
case *False*
thus *?thesis*
using *assms*
unfolding *Gauss-Jordan-column-k-det-P-def Gauss-Jordan-column-k-det-P-iarrays-def Let-def fst-conv snd-conv*
unfolding *matrix-to-iarray-nrows matrix-vector-all-zero-from-index* **by** *auto*
qed

lemma *fst-snd-Gauss-Jordan-column-k-det-P-le-nrows*:
assumes *i*: *i*≤*nrows A*
shows *fst* (*snd* (*Gauss-Jordan-column-k-det-P* (*n*, *i*, *A*) *k*)) ≤ *nrows A*
unfolding *fst-snd-Gauss-Jordan-column-k-det-P-eq-fst-snd-Gauss-Jordan-column-k-PA*[*unfolded fst-snd-Gauss-Jordan-column-k-PA-eq*]
by (*rule fst-Gauss-Jordan-column-k*[*OF i*])

The proof of the following theorem is very similar to the ones of *foldl-Gauss-Jordan-column-k-eq*, *rref-and-index-Gauss-Jordan-upt-k* and *foldl-Gauss-Jordan-column-k-det-P*.

lemma
assumes *k*<*ncols A*
shows *matrix-to-iarray-fst-Gauss-Jordan-upt-k-det-P*: *fst* (*Gauss-Jordan-upt-k-det-P A k*) = *fst* (*Gauss-Jordan-upt-k-det-P-iarrays* (*matrix-to-iarray A*) *k*)
and *matrix-to-iarray-snd-Gauss-Jordan-upt-k-det-P*: *matrix-to-iarray* (*snd* (*Gauss-Jordan-upt-k-det-P A k*)) = *snd* (*Gauss-Jordan-upt-k-det-P-iarrays* (*matrix-to-iarray A*) *k*)
and *fst* (*snd* (*foldl Gauss-Jordan-column-k-det-P* (*1*, *0*, *A*) [*0*..<*Suc k*])) ≤ *nrows A*
and *fst* (*snd* (*foldl Gauss-Jordan-column-k-det-P-iarrays* (*1*, *0*, *matrix-to-iarray A*) [*0*..<*Suc k*])) = *fst* (*snd* (*foldl Gauss-Jordan-column-k-det-P* (*1*, *0*, *A*) [*0*..<*Suc k*]))
using *assms*
proof (*induct k*)
show *fst* (*Gauss-Jordan-upt-k-det-P A 0*) = *fst* (*Gauss-Jordan-upt-k-det-P-iarrays* (*matrix-to-iarray A*) *0*)
unfolding *Gauss-Jordan-upt-k-det-P-def Gauss-Jordan-upt-k-det-P-iarrays-def Let-def fst-conv*
by (*simp*, *rule matrix-to-iarray-fst-Gauss-Jordan-column-k-det-P*, *simp-all add*: *ncols-def*)
show *matrix-to-iarray* (*snd* (*Gauss-Jordan-upt-k-det-P A 0*)) = *snd* (*Gauss-Jordan-upt-k-det-P-iarrays* (*matrix-to-iarray A*) *0*)
unfolding *Gauss-Jordan-upt-k-det-P-def Gauss-Jordan-upt-k-det-P-iarrays-def Let-def fst-conv snd-conv*
by (*simp*, *rule matrix-to-iarray-snd-snd-Gauss-Jordan-column-k-det-P*, *simp-all add*: *ncols-def*)
show *fst* (*snd* (*foldl Gauss-Jordan-column-k-det-P* (*1*, *0*, *A*) [*0*..<*Suc 0*])) ≤ *nrows A*
unfolding *Gauss-Jordan-upt-k-det-P-def Gauss-Jordan-upt-k-det-P-iarrays-def Let-def fst-conv snd-conv*
by (*simp*, *rule fst-snd-Gauss-Jordan-column-k-det-P-le-nrows*, *simp add*: *nrows-def*)
show *fst* (*snd* (*foldl Gauss-Jordan-column-k-det-P-iarrays* (*1*, *0*, *matrix-to-iarray A*) [*0*..<*Suc 0*])) = *fst* (*snd* (*foldl Gauss-Jordan-column-k-det-P* (*1*, *0*, *A*) [*0*..<*Suc 0*]))
unfolding *Gauss-Jordan-upt-k-det-P-def Gauss-Jordan-upt-k-det-P-iarrays-def Let-def fst-conv snd-conv*
by (*simp*, *rule matrix-to-iarray-fst-snd-Gauss-Jordan-column-k-det-P*[*symmetric*], *simp-all add*: *ncols-def*)
next
fix *k*

assume ($k < ncols\ A \Longrightarrow fst\ (Gauss\text{-}Jordan\text{-}upt\text{-}k\text{-}det\text{-}P\ A\ k) = fst\ (Gauss\text{-}Jordan\text{-}upt\text{-}k\text{-}det\text{-}P\text{-}iarrays\ (matrix\text{-}to\text{-}iarray\ A)\ k)$)
and ($k < ncols\ A \Longrightarrow matrix\text{-}to\text{-}iarray\ (snd\ (Gauss\text{-}Jordan\text{-}upt\text{-}k\text{-}det\text{-}P\ A\ k)) = snd\ (Gauss\text{-}Jordan\text{-}upt\text{-}k\text{-}det\text{-}P\text{-}iarrays\ (matrix\text{-}to\text{-}iarray\ A)\ k)$)
and ($k < ncols\ A \Longrightarrow fst\ (snd\ (foldl\ Gauss\text{-}Jordan\text{-}column\text{-}k\text{-}det\text{-}P\ (1,\ 0,\ A)\ [0..<Suc\ k])) \leq nrows\ A$)
and ($k < ncols\ A \Longrightarrow fst\ (snd\ (foldl\ Gauss\text{-}Jordan\text{-}column\text{-}k\text{-}det\text{-}P\text{-}iarrays\ (1,\ 0,\ matrix\text{-}to\text{-}iarray\ A)\ [0..<Suc\ k])) = fst\ (snd\ (foldl\ Gauss\text{-}Jordan\text{-}column\text{-}k\text{-}det\text{-}P\ (1,\ 0,\ A)\ [0..<Suc\ k]))$)
and *Suc-k-less-ncols*: $Suc\ k < ncols\ A$
hence *hyp1*: $fst\ (Gauss\text{-}Jordan\text{-}upt\text{-}k\text{-}det\text{-}P\ A\ k) = fst\ (Gauss\text{-}Jordan\text{-}upt\text{-}k\text{-}det\text{-}P\text{-}iarrays\ (matrix\text{-}to\text{-}iarray\ A)\ k)$
and *hyp2*: $matrix\text{-}to\text{-}iarray\ (snd\ (Gauss\text{-}Jordan\text{-}upt\text{-}k\text{-}det\text{-}P\ A\ k)) = snd\ (Gauss\text{-}Jordan\text{-}upt\text{-}k\text{-}det\text{-}P\text{-}iarrays\ (matrix\text{-}to\text{-}iarray\ A)\ k)$
and *hyp3*: $fst\ (snd\ (foldl\ Gauss\text{-}Jordan\text{-}column\text{-}k\text{-}det\text{-}P\ (1,\ 0,\ A)\ [0..<Suc\ k])) \leq nrows\ A$
and *hyp4*: $(fst\ (snd\ (foldl\ Gauss\text{-}Jordan\text{-}column\text{-}k\text{-}det\text{-}P\text{-}iarrays\ (1,\ 0,\ matrix\text{-}to\text{-}iarray\ A)\ [0..<Suc\ k])) = fst\ (snd\ (foldl\ Gauss\text{-}Jordan\text{-}column\text{-}k\text{-}det\text{-}P\ (1,\ 0,\ A)\ [0..<Suc\ k])))$
by *auto*
have *list-rw*: $[0..<Suc\ (Suc\ k)] = [0..<(Suc\ k)]\ @\ [Suc\ k]$ **by** *simp*
define *f* **where** $f = foldl\ Gauss\text{-}Jordan\text{-}column\text{-}k\text{-}det\text{-}P\ (1,\ 0,\ A)\ [0..<Suc\ k]$
define *g* **where** $g = foldl\ Gauss\text{-}Jordan\text{-}column\text{-}k\text{-}det\text{-}P\text{-}iarrays\ (1,\ 0,\ matrix\text{-}to\text{-}iarray\ A)\ [0..<Suc\ k]$
have *f-rw*: $f = (fst\ f,\ fst\ (snd\ f),\ snd\ (snd\ f))$ **by** *simp*
have *g-rw*: $g = (fst\ g,\ fst\ (snd\ g),\ snd\ (snd\ g))$ **by** *simp*
have *fst-rw*: $fst\ g = fst\ f$ **unfolding** *f-def g-def* **using** *hyp1*[*unfolded Gauss-Jordan-upt-k-det-P-def Gauss-Jordan-upt-k-det-P-iarrays-def Let-def fst-conv*] **..**
have *fst-snd-rw*: $fst\ (snd\ g) = fst\ (snd\ f)$ **unfolding** *f-def g-def* **using** *hyp4* **.**
have *snd-snd-rw*: $snd\ (snd\ g) = matrix\text{-}to\text{-}iarray\ (snd\ (snd\ f))$
unfolding *f-def g-def* **using** *hyp2*[*unfolded Gauss-Jordan-upt-k-det-P-def Gauss-Jordan-upt-k-det-P-iarrays-d*
Let-def snd-conv] **..**
have *fst-snd-f-le-nrows*: $fst\ (snd\ f) \leq nrows\ (snd\ (snd\ f))$ **unfolding** *f-def g-def*
using *hyp3* **unfolding** *nrows-def* **.**
have *Suc-k-less-ncols'*: $Suc\ k < ncols\ (snd\ (snd\ f))$ **using** *Suc-k-less-ncols* **unfolding** *ncols-def* **.**
show $fst\ (Gauss\text{-}Jordan\text{-}upt\text{-}k\text{-}det\text{-}P\ A\ (Suc\ k)) = fst\ (Gauss\text{-}Jordan\text{-}upt\text{-}k\text{-}det\text{-}P\text{-}iarrays\ (matrix\text{-}to\text{-}iarray\ A)\ (Suc\ k))$
unfolding *Gauss-Jordan-upt-k-det-P-def Gauss-Jordan-upt-k-det-P-iarrays-def Let-def fst-conv*
unfolding *list-rw foldl-append*
unfolding *List.foldl.simps*
unfolding *f-def*[*symmetric*] *g-def*[*symmetric*]
by (*subst f-rw*, *subst g-rw*, *unfold fst-rw fst-snd-rw snd-snd-rw*, *rule matrix-to-iarray-fst-Gauss-Jordan-column-k fst-snd-f-le-nrows Suc-k-less-ncols'*[])
show $matrix\text{-}to\text{-}iarray\ (snd\ (Gauss\text{-}Jordan\text{-}upt\text{-}k\text{-}det\text{-}P\ A\ (Suc\ k))) = snd\ (Gauss\text{-}Jordan\text{-}upt\text{-}k\text{-}det\text{-}P\text{-}iarrays\ (matrix\text{-}to\text{-}iarray\ A)\ (Suc\ k))$
unfolding *Gauss-Jordan-upt-k-det-P-def Gauss-Jordan-upt-k-det-P-iarrays-def Let-def snd-conv*

unfolding *list-rw foldl-append*
unfolding *List.foldl.simps*
unfolding *f-def*[*symmetric*] *g-def*[*symmetric*]
by (*subst f-rw*, *subst g-rw*, *unfold fst-rw fst-snd-rw snd-snd-rw*, *rule matrix-to-iarray-snd-snd-Gauss-Jordan-colu*
fst-snd-f-le-nrows Suc-k-less-ncols'])
show *fst* (*snd* (*foldl Gauss-Jordan-column-k-det-P* (*1*, *0*, *A*) [*0*..<*Suc* (*Suc k*)]))
≤ *nrows A*
unfolding *list-rw foldl-append List.foldl.simps*
unfolding *f-def*[*symmetric*] *g-def*[*symmetric*]
apply (*subst f-rw*)
using *fst-snd-Gauss-Jordan-column-k-det-P-le-nrows*[*OF fst-snd-f-le-nrows*]
unfolding *nrows-def* **.**
show *fst* (*snd* (*foldl Gauss-Jordan-column-k-det-P-iarrays* (*1*, *0*, *matrix-to-iarray*
A) [*0*..<*Suc* (*Suc k*)])) = *fst* (*snd* (*foldl Gauss-Jordan-column-k-det-P* (*1*, *0*, *A*)
[*0*..<*Suc* (*Suc k*)]))
unfolding *list-rw foldl-append List.foldl.simps*
unfolding *f-def*[*symmetric*] *g-def*[*symmetric*]
by (*subst f-rw*, *subst g-rw*, *unfold fst-rw fst-snd-rw snd-snd-rw*,*rule matrix-to-iarray-fst-snd-Gauss-Jordan-colun*
OF fst-snd-f-le-nrows Suc-k-less-ncols'])
qed

lemma *matrix-to-iarray-fst-Gauss-Jordan-det-P*:
shows *fst* (*Gauss-Jordan-det-P A*) = *fst* (*Gauss-Jordan-det-P-iarrays* (*matrix-to-iarray*
A))
unfolding *Gauss-Jordan-det-P-def Gauss-Jordan-det-P-iarrays-def*
using *matrix-to-iarray-fst-Gauss-Jordan-upt-k-det-P*
by (*metis diff-less matrix-to-iarray-ncols ncols-not-0 neq0-conv zero-less-one*)

lemma *matrix-to-iarray-snd-Gauss-Jordan-det-P*:
shows *matrix-to-iarray* (*snd* (*Gauss-Jordan-det-P A*)) = *snd* (*Gauss-Jordan-det-P-iarrays*
(*matrix-to-iarray A*))
unfolding *Gauss-Jordan-det-P-def Gauss-Jordan-det-P-iarrays-def*
using *matrix-to-iarray-snd-Gauss-Jordan-upt-k-det-P*
by (*metis diff-less matrix-to-iarray-ncols ncols-not-0 neq0-conv zero-less-one*)

21.3 Code equations

definition *det-iarrays A* = (*let A'* = *Gauss-Jordan-det-P-iarrays A in prod-list*
(*map* (λ*i*. (*snd A'*) !! *i* !! *i*) [*0*..<*nrows-iarray A*]) / *fst A'*)

lemma *matrix-to-iarray-det*[*code-unfold*]:
fixes *A*::'*a*::{*field*}^'*n*::{*mod-type*}^'*n*::{*mod-type*}
shows *det A* = *det-iarrays* (*matrix-to-iarray A*)
proof −
let *?f*=(λ*i*. *snd* (*Gauss-Jordan-det-P-iarrays* (*matrix-to-iarray A*)) !! *i* !! *i*)
have ∗: *fst* (*Gauss-Jordan-det-P A*) = *fst* (*Gauss-Jordan-det-P-iarrays* (*matrix-to-iarray*
A))
unfolding *matrix-to-iarray-fst-Gauss-Jordan-det-P* **..**

have *prod-list* (*map ?f* [*0..<nrows-iarray* (*matrix-to-iarray A*)]) = *prod ?f* (*set* [*0..<nrows-iarray* (*matrix-to-iarray A*)])
by (*metis* (*no-types*, *lifting*) *distinct-upt prod.distinct-set-conv-list*)
also have ... = (∏ *i*∈*UNIV*. *snd* (*Gauss-Jordan-det-P A*) $ *i* $ *i*)
proof (*rule prod.reindex-cong*[*of to-nat*::('*n*=>*nat*)])
show *inj* (*to-nat*::('*n*=>*nat*)) **by** (*metis strict-mono-imp-inj-on strict-mono-to-nat*)
show *set* [*0..<nrows-iarray* (*matrix-to-iarray A*)] = *range* (*to-nat*::'*n*=>*nat*)
unfolding *nrows-eq-card-rows* **using** *bij-to-nat*[**where** *?'a*='*n*]
unfolding *bij-betw-def*
unfolding *atLeast0LessThan atLeast-upt* **by** *auto*
fix *x*
show *snd* (*Gauss-Jordan-det-P-iarrays* (*matrix-to-iarray A*)) !! *to-nat x* !! *to-nat x* = *snd* (*Gauss-Jordan-det-P A*) $ *x* $ *x*
unfolding *matrix-to-iarray-snd-Gauss-Jordan-det-P*[*symmetric*]
unfolding *matrix-to-iarray-nth* **..**
qed
finally have (∏ *i*∈*UNIV*. *snd* (*Gauss-Jordan-det-P A*) $ *i* $ *i*)
= *prod-list* (*map* (λ*i*. *snd* (*Gauss-Jordan-det-P-iarrays* (*matrix-to-iarray A*)) !! *i* !! *i*) [*0..<nrows-iarray* (*matrix-to-iarray A*)]) **..**
thus *?thesis* **using** ∗ **unfolding** *det-code-equation det-iarrays-def Let-def* **by** *presburger*
qed

end

22 Inverse of a matrix using the Gauss Jordan algorithm over nested IArrays

theory *Inverse-IArrays*
imports
Inverse
Gauss-Jordan-PA-IArrays
begin

22.1 Definitions

definition *invertible-iarray A* = (*rank-iarray A* = *nrows-iarray A*)
definition *inverse-matrix-iarray A* = (*if invertible-iarray A then Some*(*fst*(*Gauss-Jordan-iarrays-PA A*)) *else None*)
definition *matrix-to-iarray-option A* = (*if A* ≠ *None then Some* (*matrix-to-iarray* (*the A*)) *else None*)

22.2 Some lemmas and code generation

lemma *matrix-inv-Gauss-Jordan-iarrays-PA*:
fixes *A*::'*a*::{*field*}^'*n*::{*mod-type*}^'*n*::{*mod-type*}
assumes *inv-A*: *invertible A*
shows *matrix-to-iarray* (*matrix-inv A*) = *fst* (*Gauss-Jordan-iarrays-PA* (*matrix-to-iarray A*))

by (*metis inv-A matrix-inv-Gauss-Jordan-PA matrix-to-iarray-fst-Gauss-Jordan-PA*)

lemma *matrix-to-iarray-invertible*[*code-unfold*]:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
shows *invertible A* = *invertible-iarray* (*matrix-to-iarray A*)
unfolding *invertible-iarray-def invertible-eq-full-rank*[*of A*] *matrix-to-iarray-rank matrix-to-iarray-nrows* **..**

lemma *matrix-to-iarray-option-inverse-matrix*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
shows *matrix-to-iarray-option* (*inverse-matrix A*) = (*inverse-matrix-iarray* (*matrix-to-iarray A*))
proof (*unfold inverse-matrix-def*, *auto*)
assume *inv-A*: *invertible A*
show *matrix-to-iarray-option* (*Some* (*matrix-inv A*)) = *inverse-matrix-iarray* (*matrix-to-iarray A*)
unfolding *matrix-to-iarray-option-def* **unfolding** *inverse-matrix-iarray-def* **using** *inv-A* **unfolding** *matrix-to-iarray-invertible*
using *matrix-inv-Gauss-Jordan-iarrays-PA*[*OF inv-A*] **by** *auto*
next
assume *not-inv-A*: ¬ *invertible A*
show *matrix-to-iarray-option None* = *inverse-matrix-iarray* (*matrix-to-iarray A*)
unfolding *matrix-to-iarray-option-def inverse-matrix-iarray-def*
using *not-inv-A* **unfolding** *matrix-to-iarray-invertible* **by** *simp*
qed

lemma *matrix-to-iarray-option-inverse-matrix-code*[*code-unfold*]:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
shows *matrix-to-iarray-option* (*inverse-matrix A*) = (*let matrix-to-iarray-A* = *matrix-to-iarray A*; *GJ* = *Gauss-Jordan-iarrays-PA matrix-to-iarray-A*
in if nrows-iarray matrix-to-iarray-A = *length* [*x*←*IArray.list-of* (*snd GJ*) . ¬ *is-zero-iarray x*] *then Some* (*fst GJ*) *else None*)
unfolding *matrix-to-iarray-option-inverse-matrix*
unfolding *inverse-matrix-iarray-def*
unfolding *invertible-iarray-def*
unfolding *rank-iarrays-code*
unfolding *Let-def*
unfolding *matrix-to-iarray-snd-Gauss-Jordan-PA*[*symmetric*]
unfolding *Gauss-Jordan-PA-eq*
unfolding *matrix-to-iarray-Gauss-Jordan* **by** *presburger*

lemma[*code-unfold*]:
shows *inverse-matrix-iarray A* = (*let A'* = (*Gauss-Jordan-iarrays-PA A*); *nrows* = *IArray.length A in*
(*if length* [*x*←*IArray.list-of* (*snd A'*) . ¬ *is-zero-iarray x*] = *nrows*
then Some (*fst A'*) *else None*))
unfolding *inverse-matrix-iarray-def invertible-iarray-def rank-iarrays-code Let-def*

unfolding *nrows-iarray-def snd-Gauss-Jordan-iarrays-PA-eq* **..**

end

23 Examples of computations over matrices represented as nested IArrays

theory *Examples-Gauss-Jordan-IArrays*
imports
System-Of-Equations-IArrays
Determinants-IArrays
Inverse-IArrays
Code-Z2
HOL−Library.Code-Target-Numeral

begin

23.1 Transformations between nested lists nested IArrays

definition *iarray-of-iarray-to-list-of-list* :: *'a iarray iarray => 'a list list*
where *iarray-of-iarray-to-list-of-list A = map IArray.list-of (map ((!!) A) [0..<IArray.length A])*

The following definitions are also in the file *Examples-on-Gauss-Jordan-Abstract*.

Definitions to transform a matrix to a list of list and vice versa

definition *vec-to-list* :: *'a^'n::{finite, enum} => 'a list*
where *vec-to-list A = map (($) A) (enum-class.enum::'n list)*

definition *matrix-to-list-of-list* :: *'a^'n::{finite, enum}^'m::{finite, enum} => 'a list list*
where *matrix-to-list-of-list A = map (vec-to-list) (map (($) A) (enum-class.enum::'m list))*

This definition should be equivalent to *vector-def* (in suitable types)

definition *list-to-vec* :: *'a list => 'a^'n::{enum, mod-type}*
where *list-to-vec xs = vec-lambda (% i. xs ! (to-nat i))*

lemma [*code abstract*]: *vec-nth (list-to-vec xs) = (%i. xs ! (to-nat i))*
unfolding *list-to-vec-def* **by** *fastforce*

definition *list-of-list-to-matrix* :: *'a list list => 'a^'n::{enum, mod-type}^'m::{enum, mod-type}*
where *list-of-list-to-matrix xs = vec-lambda (%i. list-to-vec (xs ! (to-nat i)))*

lemma [*code abstract*]: *vec-nth (list-of-list-to-matrix xs) = (%i. list-to-vec (xs ! (to-nat i)))*
unfolding *list-of-list-to-matrix-def* **by** *auto*

23.2 Examples

The following three lemmas are presented in both this file and in the *Examples-Gauss-Jordan-Abstract* one. They allow a more convenient printing of rational and real numbers after evaluation. They have already been added to the repository version of Isabelle, so after Isabelle2014 they should be removed from here.

lemma [*code-post*]:
int-of-integer (− *1*) = − *1*
by *simp*

lemma [*code-abbrev*]:
(*of-rat* (− *1*) :: *real*) = − *1*
by *simp*

lemma [*code-post*]:
(*of-rat* (− (*1* / *numeral k*)) :: *real*) = − *1* / *numeral k*
(*of-rat* (− (*numeral k* / *numeral l*)) :: *real*) = − *numeral k* / *numeral l*
by (*simp-all add*: *of-rat-divide of-rat-minus*)

From here on, we do the computations in two ways. The first one consists of executing the abstract functions (which internally will execute the ones over iarrays). The second one runs directly the functions over iarrays.

23.2.1 Ranks, dimensions and Gauss Jordan algorithm

In the following examples, the theorem *matrix-to-iarray-rank* (which is the file *Gauss-Jordan-IArrays* and it is a code unfold theorem) assures that the computation will be carried out using the iarrays representation.

value *vec.dim* (*col-space* (*list-of-list-to-matrix* [[*1*,*0*,*0*,*7*,*5*],[*1*,*0*,*4*,*8*,−*1*],[*1*,*0*,*0*,*9*,*8*],[*1*,*2*,*3*,*6*,*5*]]::*real*^*5*^*4*))
value *rank* (*list-of-list-to-matrix* [[*1*,*0*,*0*,*7*,*5*],[*1*,*0*,*4*,*8*,−*1*],[*1*,*0*,*0*,*9*,*8*],[*1*,*2*,*3*,*6*,*5*]]::*real*^*5*^*4*)

value *vec.dim* (*null-space* (*list-of-list-to-matrix* [[*1*,*0*,*0*,*7*,*5*],[*1*,*0*,*4*,*8*,−*1*],[*1*,*0*,*0*,*9*,*8*],[*1*,*2*,*3*,*6*,*5*]]::*rat*^*5*^*4*))

value *rank-iarray* (*IArray*[*IArray*[*1*::*rat*,*0*,*0*,*7*,*5*],*IArray*[*1*,*0*,*4*,*8*,−*1*],*IArray*[*1*,*0*,*0*,*9*,*8*],*IArray*[*1*,*2*,*3*,*6*,*5*]])

value *rank-iarray* (*IArray*[*IArray*[*1*::*real*,*0*,*1*],*IArray*[*1*,*1*,*0*],*IArray*[*0*,*1*,*1*]])
value *rank-iarray* (*IArray*[*IArray*[*1*::*bit*,*0*,*1*],*IArray*[*1*,*1*,*0*],*IArray*[*0*,*1*,*1*]])

Examples on computing the Gauss Jordan algorithm.

value *iarray-of-iarray-to-list-of-list* (*matrix-to-iarray* (*Gauss-Jordan* (*list-of-list-to-matrix* [[*Complex 1 1*,*Complex 1* (− *1*), *Complex 0 0*],[*Complex 2* (− *1*),*Complex 1 3*, *Complex 7 3*]]::*complex*^*3*^*2*)))
value *iarray-of-iarray-to-list-of-list* (*Gauss-Jordan-iarrays*(*IArray*[*IArray*[*Complex 1 1*,*Complex 1* (− *1*),*Complex 0 0*],*IArray*[*Complex 2* (− *1*),*Complex 1 3*,*Complex 7 3*]]))

23.2.2 Inverse of a matrix

Examples on inverting matrices

definition *print-result-some-iarrays A* = (*if A* = *None then None else Some* (*iarray-of-iarray-to-list-of-list* (*the A*)))

value *let A*=(*list-of-list-to-matrix* [[1,1,2,4,5,9,8],[3,0,8,4,5,0,8],[3,2,0,4,5,9,8],[3,2,8,0,5,9,8],[3,2,8,4,0
in print-result-some-iarrays (*matrix-to-iarray-option* (*inverse-matrix*
A))
value *let A*=(*IArray*[*IArray*[1::*real*,1,2,4,5,9,8],*IArray*[3,0,8,4,5,0,8],*IArray*[3,2,0,4,5,9,8],*IArray*[3,2,8,0
in print-result-some-iarrays (*inverse-matrix-iarray A*)

23.2.3 Determinant of a matrix

Examples on computing determinants of matrices

value *det* (*list-of-list-to-matrix* ([[1,8,9,1,47],[7,2,2,5,9],[3,2,7,7,4],[9,8,7,5,1],[1,2,6,4,5]])::*rat*^5^5)
value *det* (*list-of-list-to-matrix* [[1,2,7,8,9],[3,4,12,10,7],[−5,4,8,7,4],[0,1,2,4,8],[9,8,7,13,11]]::*rat*^5^5)

value *det-iarrays* (*IArray*[*IArray*[1::*real*,2,7,8,9],*IArray*[3,4,12,10,7],*IArray*[−5,4,8,7,4],*IArray*[0,1,2,4,8],
value *det-iarrays* (*IArray*[*IArray*[286,662,263,246,642,656,351,454,339,848],
IArray[307,489,667,908,103,47,120,133,85,834],
IArray[69,732,285,147,527,655,732,661,846,202],
IArray[463,855,78,338,786,954,593,550,913,378],
IArray[90,926,201,362,985,341,540,912,494,427],
IArray[384,511,12,627,131,620,987,996,445,216],
IArray[385,538,362,643,567,804,499,914,332,512],
IArray[879,159,312,187,827,503,823,893,139,546],
IArray[800,376,331,363,840,737,911,886,456,848],
IArray[900,737,280,370,121,195,958,862,957,754::*real*]])

23.2.4 Bases of the fundamental subspaces

Examples on computing basis for null space, row space, column space and left null space.

value *let A* = (*list-of-list-to-matrix* ([[1,3,−2,0,2,0],[2,6,−5,−2,4,−3],[0,0,5,10,0,15],[2,6,0,8,4,18]])::*real*^

in vec-to-list‘ (*basis-null-space A*)
value *let A* = (*list-of-list-to-matrix* ([[3,4,0,7],[1,−5,2,−2],[−1,4,0,3],[1,−1,2,2]])::*real*^4^4)

in vec-to-list‘ (*basis-null-space A*)

value *let A* = (*IArray*[*IArray*[1::*real*,3,−2,0,2,0],*IArray*[2,6,−5,−2,4,−3],*IArray*[0,0,5,10,0,15],*IArray*[2,6

in IArray.list-of‘ (*basis-null-space-iarrays A*)
value *let A* = (*IArray*[*IArray*[3::*real*,4,0,7],*IArray*[1,−5,2,−2],*IArray*[−1,4,0,3],*IArray*[1,−1,2,2]])
in IArray.list-of‘ (*basis-null-space-iarrays A*)

value *let A = (list-of-list-to-matrix ([[1,3,−2,0,2,0],[2,6,−5,−2,4,−3],[0,0,5,10,0,15],[2,6,0,8,4,18]])::real*

in vec-to-list‘ (basis-row-space A)
value *let A = (list-of-list-to-matrix ([[3,4,0,7],[1,−5,2,−2],[−1,4,0,3],[1,−1,2,2]])::real^4^4)*

in vec-to-list‘ (basis-row-space A)

value *let A = (IArray[IArray[1::real,3,−2,0,2,0],IArray[2,6,−5,−2,4,−3],IArray[0,0,5,10,0,15],IArray[2,6*

in IArray.list-of‘ (basis-row-space-iarrays A)
value *let A = (IArray[IArray[3::real,4,0,7],IArray[1,−5,2,−2],IArray[−1,4,0,3],IArray[1,−1,2,2]])*
in IArray.list-of‘ (basis-row-space-iarrays A)

value *let A = (list-of-list-to-matrix ([[1,3,−2,0,2,0],[2,6,−5,−2,4,−3],[0,0,5,10,0,15],[2,6,0,8,4,18]])::real*

in vec-to-list‘ (basis-col-space A)
value *let A = (list-of-list-to-matrix ([[3,4,0,7],[1,−5,2,−2],[−1,4,0,3],[1,−1,2,2]])::real^4^4)*

in vec-to-list‘ (basis-col-space A)

value *let A = (IArray[IArray[1::real,3,−2,0,2,0],IArray[2,6,−5,−2,4,−3],IArray[0,0,5,10,0,15],IArray[2,6*

in IArray.list-of‘ (basis-col-space-iarrays A)
value *let A = (IArray[IArray[3::real,4,0,7],IArray[1,−5,2,−2],IArray[−1,4,0,3],IArray[1,−1,2,2]])*
in IArray.list-of‘ (basis-col-space-iarrays A)

value *let A = (list-of-list-to-matrix ([[1,3,−2,0,2,0],[2,6,−5,−2,4,−3],[0,0,5,10,0,15],[2,6,0,8,4,18]])::real*

in vec-to-list‘ (basis-left-null-space A)
value *let A = (list-of-list-to-matrix ([[3,4,0,7],[1,−5,2,−2],[−1,4,0,3],[1,−1,2,2]])::real^4^4)*

in vec-to-list‘ (basis-left-null-space A)

value *let A = (IArray[IArray[1::real,3,−2,0,2,0],IArray[2,6,−5,−2,4,−3],IArray[0,0,5,10,0,15],IArray[2,6*

in IArray.list-of‘ (basis-left-null-space-iarrays A)
value *let A = (IArray[IArray[3::real,4,0,7],IArray[1,−5,2,−2],IArray[−1,4,0,3],IArray[1,−1,2,2]])*
in IArray.list-of‘ (basis-left-null-space-iarrays A)

23.2.5 Consistency and inconsistency

Examples on checking the consistency/inconsistency of a system of equations. The theorems *matrix-to-iarray-independent-and-consistent* and *matrix-to-iarray-dependent-and-consistent* which are code theorems and they are in the file *System-Of-Equations-IArrays* assure the execution using the iarrays representation.

value *independent-and-consistent* (*list-of-list-to-matrix* ([[*1*,*0*,*0*],[*0*,*1*,*0*],[*0*,*0*,*1*],[*0*,*0*,*0*],[*0*,*0*,*0*]])::*real^3^5*)
(*list-to-vec*([*2*,*3*,*4*,*0*,*0*])::*real^5*)
value *consistent* (*list-of-list-to-matrix* ([[*1*,*0*,*0*],[*0*,*1*,*0*],[*0*,*0*,*1*],[*0*,*0*,*0*],[*0*,*0*,*0*]])::*real^3^5*)
(*list-to-vec*([*2*,*3*,*4*,*0*,*0*])::*real^5*)
value *inconsistent* (*list-of-list-to-matrix* ([[*1*,*0*,*0*],[*0*,*1*,*0*],[*3*,*0*,*1*],[*0*,*7*,*0*],[*0*,*0*,*9*]])::*real^3^5*)
(*list-to-vec*([*2*,*0*,*4*,*0*,*0*])::*real^5*)
value *dependent-and-consistent* (*list-of-list-to-matrix* ([[*1*,*0*,*0*],[*0*,*1*,*0*]])::*real^3^2*)
(*list-to-vec*([*3*,*4*])::*real^2*)
value *independent-and-consistent* (*mat 1*::*real^3^3*) (*list-to-vec*([*3*,*4*,*5*])::*real^3*)

23.2.6 Solving systems of linear equations

Examples on solving linear systems.

definition *print-result-system-iarrays A* = (*if A* = *None then None else Some*
(*IArray.list-of* (*fst* (*the A*)), *IArray.list-of'* (*snd* (*the A*))))

value *let A* = (*list-of-list-to-matrix* [[*0*,*0*,*0*],[*0*,*0*,*0*],[*0*,*0*,*1*]]::*real^3^3*); *b*=(*list-to-vec*
[*4*,*5*,*0*]::*real^3*);
result = *pair-vec-vecset* (*solve A b*)
in print-result-system-iarrays (*result*)
value *let A* = (*list-of-list-to-matrix* [[*3*,*2*,*5*,*2*,*7*],[*6*,*4*,*7*,*4*,*5*],[*3*,*2*,−*1*,*2*,−*11*],[*6*,*4*,*1*,*4*,−*13*]]::*real^5^4*);
b=(*list-to-vec* [*0*,*0*,*0*,*0*]::*real^4*);
result = *pair-vec-vecset* (*solve A b*)
in print-result-system-iarrays (*result*)
value *let A* = (*list-of-list-to-matrix* [[*4*,*5*,*8*],[*9*,*8*,*7*],[*4*,*6*,*1*]]::*real^3^3*); *b*=(*list-to-vec*
[*4*,*5*,*8*]::*real^3*);
result = *pair-vec-vecset* (*solve A b*)
in print-result-system-iarrays (*result*)

value *let A* = (*IArray*[*IArray*[*0*::*real*,*0*,*0*],*IArray*[*0*,*0*,*0*],*IArray*[*0*,*0*,*1*]]); *b*=(*IArray*[*4*,*5*,*0*]);
result = (*solve-iarrays A b*)
in print-result-system-iarrays (*result*)
value *let A* = (*IArray*[*IArray*[*3*::*real*,*2*,*5*,*2*,*7*],*IArray*[*6*,*4*,*7*,*4*,*5*],*IArray*[*3*,*2*,−*1*,*2*,−*11*],*IArray*[*6*,*4*,*1*,*4*,−*13*]])
b=(*IArray*[*0*,*0*,*0*,*0*]);
result = (*solve-iarrays A b*)
in print-result-system-iarrays (*result*)
value *let A* = (*IArray*[*IArray*[*4*,*5*,*8*],*IArray*[*9*::*real*,*8*,*7*],*IArray*[*4*,*6*::*real*,*1*]]); *b*=(*IArray*[*4*,*5*,*8*]);
result = (*solve-iarrays A b*)
in print-result-system-iarrays (*result*)

export-code
rank-iarray
inverse-matrix-iarray
det-iarrays
consistent-iarrays
inconsistent-iarrays
independent-and-consistent-iarrays
dependent-and-consistent-iarrays
basis-left-null-space-iarrays

basis-null-space-iarrays
basis-col-space-iarrays
basis-row-space-iarrays
solve-iarrays
in *SML*
end

24 Exporting code to SML and Haskell

theory *Code-Generation-IArrays*
imports
Examples-Gauss-Jordan-IArrays
begin

24.1 Exporting code

The following two equations are necessary to execute code. If we don't remove them from code unfold, the exported code will not work (there exist problems with the number 1 and number 0. Those problems appear when the HMA library is imported).

lemma [*code-unfold del*]: *1* ≡ *real-of-rat 1* **by** *simp*
lemma [*code-unfold del*]: *0* ≡ *real-of-rat 0* **by** *simp*

definition *matrix-z2* = *IArray*[*IArray*[*0*,*1*], *IArray*[*1*,*1*::*bit*], *IArray*[*1*,*0*::*bit*]]
definition *matrix-rat* = *IArray*[*IArray*[*1*,*0*,*8*], *IArray*[*5.7*,*22*,*1*], *IArray*[*41*,−*58*/*7*,*78*::*rat*]]
definition *matrix-real* = *IArray*[*IArray*[*0*,*1*], *IArray*[*1*,−*2*::*real*]]
definition *vec-rat* = *IArray*[*21*,*5*,*7*::*rat*]

definition *print-result-Gauss A* = *iarray-of-iarray-to-list-of-list* (*Gauss-Jordan-iarrays A*)
definition *print-rank A* = *rank-iarray A*
definition *print-det A* = *det-iarrays A*

definition *print-result-z2* = *print-result-Gauss* (*matrix-z2*)
definition *print-result-rat* = *print-result-Gauss* (*matrix-rat*)
definition *print-result-real* = *print-result-Gauss* (*matrix-real*)

definition *print-rank-z2* = *print-rank* (*matrix-z2*)
definition *print-rank-rat* = *print-rank* (*matrix-rat*)
definition *print-rank-real* = *print-rank* (*matrix-real*)

definition *print-det-rat* = *print-det* (*matrix-rat*)
definition *print-det-real* = *print-det* (*matrix-real*)

definition *print-inverse A* = *inverse-matrix-iarray A*
definition *print-inverse-real A* = *print-inverse* (*matrix-real*)
definition *print-inverse-rat A* = *print-inverse* (*matrix-rat*)

definition *print-system A b = print-result-system-iarrays* (*solve-iarrays A b*)
definition *print-system-rat = print-result-system-iarrays* (*solve-iarrays matrix-rat vec-rat*)

24.2 The Mathematica bug

Varona et al [1] detected that the commercial software *Mathematica*®, in its versions 8.0, 9.0 and 9.0.1, was computing erroneously determinants of matrices of big integers, even for small dimensions (in their work they present an example of a matrix of dimension 14×14). The situation is such that even the same determinant, computed twice, produces two different and wrong results.

Here we reproduce that example. The computation of the determinant is correctly carried out by the exported code from this formalization, in both SML and Haskell.

definition *powersMatrix =*
IArray[
IArray[*10^123*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*],
IArray[*0*,*10^152*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*],
IArray[*0*,*0*,*10^185*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*],
IArray[*0*,*0*,*0*,*10^220*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*],
IArray[*0*,*0*,*0*,*0*,*10^397*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*],
IArray[*0*,*0*,*0*,*0*,*0*,*10^449*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*],
IArray[*0*,*0*,*0*,*0*,*0*,*0*,*10^503*,*0*,*0*,*0*,*0*,*0*,*0*,*0*],
IArray[*0*,*0*,*0*,*0*,*0*,*0*,*0*,*10^563*,*0*,*0*,*0*,*0*,*0*,*0*],
IArray[*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*10^979*,*0*,*0*,*0*,*0*,*0*],
IArray[*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*10^1059*,*0*,*0*,*0*,*0*],
IArray[*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*10^1143*,*0*,*0*,*0*],
IArray[*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*10^1229*,*0*,*0*],
IArray[*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*10^1319*,*0*],
IArray[*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,(*10^1412*)::*rat*]]

definition *basicMatrix =*
IArray[
IArray[*−32*, *69*, *89*,*−60*,*−83*,*−22*,*−14*,*−58*,*85*,*56*,*−65*,*−30*,*−86*,*−9*],
IArray[*6*, *99*, *11*, *57*, *47*,*−42*,*−48*,*−65*, *25*, *50*,*−70*,*−3*,*−90*, *31*],
IArray[*78*, *38*, *12*, *64*,*−67*,*−4*,*−52*,*−65*, *19*, *71*, *38*,*−17*, *51*,*−3*],
IArray[*−93*, *30*, *89*, *22*, *13*, *48*,*−73*, *93*, *11*,*−97*,*−49*, *61*,*−25*,*−4*],
IArray[*54*,*−22*, *54*,*−53*,*−52*, *64*, *19*, *1*, *81*,*−72*,*−11*, *50*, *0*,*−81*],
IArray[*65*,*−58*, *3*, *57*, *19*, *77*, *76*,*−57*,*−80*, *22*, *93*,*−85*, *67*, *58*],
IArray[*29*,*−58*, *47*, *87*, *3*,*−6*,*−81*, *5*, *98*, *86*,*−98*, *51*,*−62*,*−66*],
IArray[*93*,*−77*, *16*,*−64*, *48*, *84*, *97*, *75*, *89*, *63*, *34*,*−98*,*−94*, *19*],
IArray[*45*,*−99*, *3*,*−57*, *32*, *60*, *74*, *4*, *69*, *98*,*−40*,*−69*,*−28*,*−26*],
IArray[*−13*, *51*,*−99*,*−2*, *48*, *71*,*−81*,*−32*, *78*, *27*,*−28*,*−22*, *22*, *94*],
IArray[*11*, *72*,*−74*, *86*, *79*,*−58*,*−89*, *80*, *70*, *55*,*−49*, *51*,*−42*, *66*],
IArray[*−72*, *53*, *49*,*−46*, *17*,*−22*,*−48*,*−40*,*−28*,*−85*, *88*,*−30*, *74*, *32*],
IArray[*−92*,*−22*,*−90*, *67*,*−25*,*−28*,*−91*,*−8*, *32*,*−41*, *10*, *6*, *85*, *21*],

IArray[*47*,−*73*,−*30*,−*60*, *99*, *9*,−*86*,−*70*, *84*, *55*, *19*, *69*, *11*,−*84*::*rat*]]

definition *smallMatrix*=
IArray[
IArray[*528*, *853*,−*547*,−*323*, *393*,−*916*,−*11*,−*976*, *279*,−*665*, *906*,−*277*, *103*,−*485*],
IArray[*878*, *910*,−*306*,−*260*, *575*,−*765*,−*32*, *94*, *254*, *276*,−*156*, *625*,−*8*,−*566*],
IArray[−*357*, *451*,−*475*, *327*,−*84*, *237*, *647*, *505*,−*137*, *363*,−*808*, *332*, *222*,−*998*],
IArray[−*76*, *26*,−*778*, *505*, *942*,−*561*,−*350*, *698*,−*532*,−*507*,−*78*,−*758*, *346*,−*545*],
IArray[−*358*, *18*,−*229*,−*880*,−*955*,−*346*, *550*,−*958*, *867*,−*541*,−*962*, *646*, *932*,
168],
IArray[*192*, *233*, *620*,*955*,−*877*, *281*, *357*,−*226*,−*820*, *513*,−*882*, *536*,−*237*, *877*],
IArray[−*234*,−*71*,−*831*, *880*,−*135*,−*249*,−*427*, *737*, *664*, *298*,−*552*,−*1*,−*712*,−*691*],
IArray[*80*, *748*, *684*, *332*, *730*,−*111*,−*643*, *102*,−*242*,−*82*,−*28*, *585*, *207*,−*986*],
IArray[*967*, *1*,−*494*, *633*, *891*,−*907*,−*586*, *129*, *688*, *150*,−*501*,−*298*, *704*,−*68*],
IArray[*406*,−*944*,−*533*,−*827*, *615*, *907*,−*443*,−*350*, *700*,−*878*, *706*, *1*,*800*, *120*],
IArray[*33*,−*328*,−*543*, *583*,−*443*,−*635*,*904*,−*745*,−*398*,−*110*, *751*, *660*, *474*, *255*],
IArray[−*537*,−*311*, *829*, *28*, *175*, *182*,−*930*, *258*,−*808*,−*399*,−*43*,−*68*,−*553*, *421*],
IArray[−*373*,−*447*,−*252*,−*619*,−*418*, *764*, *994*,−*543*,−*37*,−*845*, *30*,−*704*, *147*,−*534*],
IArray[*638*,−*33*, *932*,−*335*,−*75*,−*676*,−*934*, *239*, *210*, *665*, *414*,−*803*, *564*,−*805*::*rat*]]

definition *bigMatrix*=(*basicMatrix* ∗∗*i* *powersMatrix*) + *smallMatrix*

end

25 Exporting code to SML

theory *Code-Generation-IArrays-SML*
imports
HOL−Library.Code-Real-Approx-By-Float
Code-Generation-IArrays
begin

Some serializations of gcd and abs in SML. Since gcd is not part of the standard SML library, we have serialized it to the corresponding operation in PolyML and MLton.

context
includes *integer.lifting*
begin

lift-definition *gcd-integer* :: *integer* => *integer* => *integer*
is *gcd* :: *int* => *int* => *int* **.**

lemma *gcd-integer-code*[*code*]:
gcd-integer l k = |*if l* = (*0*::*integer*) *then k else gcd-integer l* (|*k*| *mod* |*l*|)|
apply (*transfer*) **using** *gcd-code-int* **by** (*metis gcd.commute*)
end

code-printing
constant *abs* :: *integer* => - ⇀ (*SML*) *IntInf.abs*
| **constant** *gcd-integer* :: *integer* => - => - ⇀ (*SML*) (*PolyML.IntInf.gcd* ((-),(-)))

lemma *gcd-code*[*code*]:
gcd a b = *int-of-integer* (*gcd-integer* (*of-int a*) (*of-int b*))
by (*metis gcd-integer.abs-eq int-of-integer-integer-of-int integer-of-int-eq-of-int*)

code-printing
constant *abs* :: *real* => *real* ⇀
(*SML*) *Real.abs*

declare [[*code drop*: *abs* :: *real* ⇒ *real*]]

There are several ways to serialize div and mod. The following ones are four examples of it:

code-printing
constant *divmod-integer* :: *integer* => - => - ⇀ (*SML*) (*IntInf.quotRem* ((-),(-)))

export-code
print-rank-real
print-rank-rat
print-rank-z2
print-rank
print-result-real
print-result-rat
print-result-z2
print-result-Gauss
print-det-rat
print-det-real
print-det
print-inverse-real
print-inverse-rat
print-inverse
print-system-rat
print-system
in *SML* **module-name** *Gauss-SML*

end

26 Code Generation for rational numbers in Haskell

theory *Code-Rational*

imports

HOL−Library.Code-Real-Approx-By-Float
Code-Generation-IArrays

HOL−Library.Code-Target-Int
begin

26.1 Serializations

The following *code-printing code-module* module is the usual way to import libraries in Haskell. In this case, we rebind some functions from Data.Ratio. See https://lists.cam.ac.uk/pipermail/cl-isabelle-users/2013-June/msg00007.html

code-printing code-module *Rational* ⇀ (*Haskell*)
‹*module Rational*(*fract*, *numerator*, *denominator*) *where*

import qualified Data.Ratio
import Data.Ratio(*numerator*, *denominator*)

fract (*a*, *b*) = *a Data.Ratio.*% *b*›

context
includes *integer.lifting*
begin
lift-definition *Frct-integer* :: *integer* × *integer* ⇒ *rat*
is *Frct* :: *int* × *int* ⇒ *rat* **.**
end

consts *Foo*::*rat*
code-datatype *Foo*

context
includes *integer.lifting*
begin

lemma [*code*]:
Frct a = *Frct-integer* ((*of-int* (*fst a*)), (*of-int* (*snd a*)))
by (*transfer*, *simp*)

lemma [*code*]:
Rat.of-int a = *Frct-integer* (*of-int a*, *1*)
by *transfer* (*auto simp*: *Fract-of-int-eq Rat.of-int-def*)

definition *numerator* :: *rat* ⇒ *int*

where *numerator x = fst (quotient-of x)*

definition *denominator* :: *rat => int*
where *denominator x = snd (quotient-of x)*

lift-definition *numerator-integer* :: *rat => integer*
is *numerator* **.**

lift-definition *denominator-integer* :: *rat => integer*
is *denominator* **.**

lemma [*code*]:
inverse x = Frct-integer (denominator-integer x, numerator-integer x)
apply (*cases x*)
apply *transfer*
apply (*auto simp*: *inverse-eq-divide numerator-def denominator-def quotient-of-Fract One-rat-def*)
done

lemma *quotient-of-num-den*: *quotient-of x = ((numerator x), (denominator x))*
unfolding *numerator-def denominator-def*
by *simp*

lemma [*code*]: *quotient-of x = (int-of-integer (numerator-integer x), int-of-integer(denominator-integer x))*
by (*transfer*, *simp add*: *quotient-of-num-den*)
end

code-printing
type-constructor *rat* ⇀ (*Haskell*) *Prelude.Rational*
| **class-instance** *rat* :: *HOL.equal* => (*Haskell*) −
| **constant** *0* :: *rat* ⇀ (*Haskell*) (*Prelude.toRational* (*0*::*Integer*))
| **constant** *1* :: *rat* ⇀ (*Haskell*) (*Prelude.toRational* (*1*::*Integer*))
| **constant** *Frct-integer* ⇀ (*Haskell*) *Rational.fract* (-)
| **constant** *numerator-integer* ⇀ (*Haskell*) *Rational.numerator* (-)
| **constant** *denominator-integer* ⇀ (*Haskell*) *Rational.denominator* (-)
| **constant** *HOL.equal* :: *rat* ⇒ *rat* ⇒ *bool* ⇀
(*Haskell*) (-) == (-)
| **constant** (<) :: *rat* => *rat* => *bool* ⇀
(*Haskell*) - < -
| **constant** (≤) :: *rat* => *rat* => *bool* ⇀
(*Haskell*) - <= -
| **constant** (+) :: *rat* ⇒ *rat* ⇒ *rat* ⇀
(*Haskell*) (-) + (-)
| **constant** (−) :: *rat* ⇒ *rat* ⇒ *rat* ⇀
(*Haskell*) (-) − (-)
| **constant** (∗) :: *rat* ⇒ *rat* ⇒ *rat* ⇀
(*Haskell*) (-) ∗ (-)

| **constant** (/) :: *rat* ⇒ *rat* ⇒ *rat* ⇀
(*Haskell*) (-) *′*/ (-)
| **constant** *uminus* :: *rat* => *rat* ⇀
(*Haskell*) *Prelude.negate*

end

27 Exporting code to Haskell

theory *Code-Generation-IArrays-Haskell*
imports
Code-Rational
begin

export-code
print-rank-real
print-rank-rat
print-rank-z2
print-rank
print-result-real
print-result-rat
print-result-z2
print-result-Gauss
print-det-rat
print-det-real
print-det
print-inverse-real
print-inverse-rat
print-inverse
print-system-rat
print-system
in *Haskell*
module-name *Gauss-Haskell*

end

References

[1] M. P. Antonio J. Durán and J. L. Varona. Misfortunes of a mathematicians' trio using computer algebra systems: Can we trust? *CoRR*, abs/1312.3270, 2013. http://arxiv.org/abs/1312.3270.