

QR Decomposition

By Jose Divasón and Jesús Aransay*

December 14, 2021

Abstract

In this work we present a formalization of the QR decomposition, an algorithm which decomposes a real matrix A in the product of another two matrices Q and R , where Q is an orthogonal matrix and R is invertible and upper triangular. The algorithm is useful for the least squares problem, i.e. the computation of the best approximation of an unsolvable system of linear equations. As a side-product, the Gram-Schmidt process has also been formalized. A refinement using immutable arrays is presented as well. The development relies, among others, on the AFP entry *Implementing field extensions of the form $\mathbb{Q}[\sqrt{b}]$* by René Thiemann, which allows to execute the algorithm using symbolic computations. Verified code can be generated and executed using floats as well.

Contents

1	Miscellaneous file for the QR algorithm	1
2	Projections	10
2.1	Definitions of vector projection and projection of a vector onto a set.	10
2.2	Properties	10
2.3	Orthogonal Complement	11
2.4	Normalization of vectors	13
3	The Gram-Schmidt algorithm	14
3.1	Gram-Schmidt algorithm	14
3.1.1	First way	15
3.1.2	Second way	18
3.1.3	Third way	19
3.1.4	Examples of execution	34

*This research has been funded by the research grant FPI-UR-12 of the Universidad de La Rioja and by the project MTM2014-54151-P from Ministerio de Economía y Competitividad (Gobierno de España).

4	QR Decomposition	35
4.1	The QR Decomposition of a matrix	35
4.1.1	Divide a vector by its norm	36
4.1.2	The QR Decomposition	38
5	Least Squares Approximation	53
5.1	Second part of the Fundamental Theorem of Linear Algebra .	53
5.2	Least Squares Approximation	53
6	Examples of execution using floats	63
6.0.1	Examples	63
7	Examples of execution using symbolic computation	63
7.1	Execution of the QR decomposition using symbolic computation	64
7.1.1	Some previous definitions and lemmas	64
7.1.2	Examples	64
8	IArray Addenda QR	65
8.1	Some previous instances	66
8.2	Some previous definitions and properties for IArrays	69
8.2.1	Lemmas	69
8.2.2	Definitions	69
8.3	Code generation	69
9	Matrices as nested IArrays	69
9.1	Isomorphism between matrices implemented by vecs and matrices implemented by iarrays	70
9.1.1	Isomorphism between vec and iarray	70
9.1.2	Isomorphism between matrix and nested iarrays	71
9.2	Definition of operations over matrices implemented by iarrays	72
9.2.1	Properties of previous definitions	73
9.3	Definition of elementary operations	75
9.3.1	Code generator	76
10	Gram Schmidt over IArrays	79
10.1	Some previous definitions, lemmas and instantiations about iarrays	79
10.2	Inner mult over real iarrays	83
10.3	Gram Schmidt over IArrays	84
11	QR Decomposition over iarrays	91
11.1	QR Decomposition refinement over iarrays	91
12	Examples of execution using floats and IArrays	93
12.1	Examples	94

13	Examples of execution using symbolic computation and iarrays	95
13.1	Execution of the QR decomposition using symbolic computation and iarrays	95
13.1.1	Examples	96
14	Generalization of the Second Part of the Fundamental Theorem of Linear Algebra	99
14.1	Conjugate class	99
14.2	Real_of_extended class	100
14.3	Generalizing HMA	101
14.3.1	Inner product spaces	101
14.3.2	Orthogonality	105
14.4	Vecs as inner product spaces	106
14.5	Matrices and inner product	108
14.6	Orthogonal complement generalized	108
14.7	Generalizing projections	109
14.8	Second Part of the Fundamental Theorem of Linear Algebra generalized	110
15	Improvements to get better performance of the algorithm	112
15.1	Improvements for computing the Gram Schmidt algorithm and QR decomposition using vecs	112
15.1.1	New definitions	112
15.1.2	General properties about <i>sum-list</i>	112
15.1.3	Proving a code equation to improve the performance	114
15.2	Improvements for computing the Gram Schmidt algorithm and QR decomposition using immutable arrays	120
15.2.1	New definitions	120
15.2.2	General properties	120
15.2.3	Proving the equivalence	122
15.3	Other code equations that improve the performance	127

1 Miscellaneous file for the QR algorithm

```
theory Miscellaneous-QR
imports
  Gauss-Jordan.Examples-Gauss-Jordan-Abstract
begin
```

These two lemmas maybe should be in the file *Code-Matrix.thy* of the Gauss-Jordan development.

```
lemma [code abstract]: vec-nth (a - b) = (%i. a$i - b$i) by (metis vector-minus-component)
lemma [code abstract]: vec-nth (c *_R x) = (%i. c *_R (x$i)) by auto
```

This lemma maybe should be in the file *Mod-Type.thy* of the Gauss-Jordan development.

```

lemma from-nat-le:
  fixes  $i::'a::\{\text{mod-type}\}$ 
  assumes  $i: \text{to-nat } i < k$ 
  and  $k: k < \text{CARD}('a)$ 
  shows  $i < \text{from-nat } k$ 
  by (metis (full-types) from-nat-mono from-nat-to-nat-id i k)

```

Some properties about orthogonal matrices.

```

lemma orthogonal-mult:
  assumes orthogonal a b
  shows orthogonal (x *R a) (y *R b)
  using assms unfolding orthogonal-def by simp

```

```

lemma orthogonal-matrix-is-orthogonal:
  fixes  $A::\text{real}^n{}^n$ 
  assumes  $o: \text{orthogonal-matrix } A$ 
  shows (pairwise orthogonal (columns A))
proof (unfold pairwise-def columns-def, auto)
  fix  $i j$ 
  assume column-i-not-j: column i A  $\neq$  column j A
  hence i-not-j: i  $\neq$  j by auto
  have  $0 = (\text{mat } 1) \$ i \$ j$  by (metis i-not-j mat-1-fun)
  also have  $\dots = (\text{transpose } A ** A) \$ i \$ j$  using  $o$  unfolding orthogonal-matrix
by simp
  also have  $\dots = \text{row } i (\text{transpose } A) \cdot \text{column } j A$  unfolding matrix-matrix-mult-inner-mult
by simp
  also have  $\dots = \text{column } i A \cdot \text{column } j A$  unfolding row-transpose ..
  finally show orthogonal (column i A) (column j A) unfolding orthogonal-def ..
qed

```

```

lemma orthogonal-matrix-norm:
  fixes  $A::\text{real}^n{}^n$ 
  assumes  $o: \text{orthogonal-matrix } A$ 
  shows  $\text{norm } (\text{column } i A) = 1$ 
proof –
  have  $1 = (\text{transpose } A ** A) \$ i \$ i$  using  $o$  unfolding orthogonal-matrix by
(simp add: mat-1-fun)
  also have  $\dots = (\text{column } i A) \cdot (\text{column } i A)$  unfolding matrix-matrix-mult-inner-mult
row-transpose ..
  finally show  $\text{norm } (\text{column } i A) = 1$  using norm-eq-1 by auto
qed

```

```

lemma orthogonal-matrix-card:
  fixes  $A::\text{real}^n{}^n$ 
  assumes  $o: \text{orthogonal-matrix } A$ 
  shows  $\text{card } (\text{columns } A) = \text{ncols } A$ 
proof (rule ccontr)

```

```

assume card-not-ncols:  $\text{card}(\text{columns } A) \neq \text{ncols } A$ 
have  $\exists i j. \text{column } i A = \text{column } j A \wedge i \neq j$ 
proof (rule ccontr, auto)
  assume col-eq:  $\forall i j. \text{column } i A = \text{column } j A \longrightarrow i = j$ 
  have  $\text{card}(\text{columns } A) = \text{card} \{i. i \in (\text{UNIV}::'n \text{ set})\}$ 
    by (rule bij-betw-same-card[symmetric, of  $\lambda i. \text{column } i A$ ],
      auto simp add: bij-betw-def columns-def inj-on-def col-eq)
  also have ... =  $\text{ncols } A$  unfolding ncols-def by simp
  finally show False using card-not-ncols by contradiction
qed
from this obtain  $i j$  where col-eq:  $\text{column } i A = \text{column } j A$  and i-not-j:  $i \neq j$ 
by auto
  have  $0 = (\text{mat } 1) \$ i \$ j$  using mat-1-fun i-not-j by metis
  also have ... =  $(\text{transpose } A ** A) \$ i \$ j$  using o unfolding orthogonal-matrix
by simp
  also have ... =  $\text{column } i A \cdot \text{column } j A$  unfolding matrix-matrix-mult-inner-mult
row-transpose ..
  show False
    by (metis calculation col-eq mat-1-fun matrix-matrix-mult-inner-mult
      o orthogonal-matrix zero-neq-one)
qed

```

lemma *orthogonal-matrix-intro*:

```

fixes  $A::\text{real}^n{}^n$ 
assumes p: (pairwise orthogonal (columns  $A$ ))
and  $n$ :  $\forall i. \text{norm}(\text{column } i A) = 1$ 
and  $c$ :  $\text{card}(\text{columns } A) = \text{ncols } A$ 
shows orthogonal-matrix  $A$ 
proof (unfold orthogonal-matrix vec-eq-iff, clarify, unfold mat-1-fun, auto)
  fix  $ia$ 
  have  $(\text{transpose } A ** A) \$ ia \$ ia = \text{column } ia A \cdot \text{column } ia A$ 
    unfolding matrix-matrix-mult-inner-mult unfolding row-transpose ..
  also have ... = 1 using  $n$  norm-eq-1 by blast
  finally show  $(\text{transpose } A ** A) \$ ia \$ ia = 1$  .
  fix  $i$ 
  assume i-not-ia:  $i \neq ia$ 
  have column-i-not-ia:  $\text{column } i A \neq \text{column } ia A$ 
  proof (rule ccontr, simp)
    assume col-i-ia:  $\text{column } i A = \text{column } ia A$ 
    have  $rw: (\lambda i. \text{column } i A)'(\text{UNIV} - \{ia\}) = \{\text{column } i A \mid i. i \neq ia\}$  unfolding
columns-def by auto
    have  $\text{card}(\text{columns } A) = \text{card}(\{\text{column } i A \mid i. i \neq ia\})$ 
      by (rule bij-betw-same-card[of id], unfold bij-betw-def columns-def)
      (auto, metis col-i-ia i-not-ia)
    also have ... =  $\text{card}((\lambda i. \text{column } i A)'(\text{UNIV} - \{ia\}))$  unfolding rw ..
    also have ...  $\leq \text{card}(\text{UNIV} - \{ia\})$  by (metis card-image-le finite-code)
    also have ...  $< \text{CARD } ('n)$  by simp
    finally show False using  $c$  unfolding ncols-def by simp

```

qed
hence *oia*: *orthogonal* (column *i* *A*) (column *ia* *A*)
using *p* **unfolding** *pairwise-def* **unfolding** *columns-def* **by** *auto*
have (*transpose* *A* ** *A*) \$ *i* \$ *ia* = column *i* *A* · column *ia* *A*
unfolding *matrix-matrix-mult-inner-mult* **unfolding** *row-transpose* ..
also have ... = 0 **using** *oia* **unfolding** *orthogonal-def* .
finally show (*transpose* *A* ** *A*) \$ *i* \$ *ia* = 0 .
qed

lemma *orthogonal-matrix2*:
fixes *A*::*real*^{*n*}^{*n*}
shows *orthogonal-matrix* *A* = ((*pairwise orthogonal* (columns *A*)) ∧ (∀ *i*. *norm*
(column *i* *A*) = 1) ∧
(*card* (columns *A*) = *ncols* *A*))
using *orthogonal-matrix-intro*[of *A*]
orthogonal-matrix-is-orthogonal[of *A*]
orthogonal-matrix-norm[of *A*]
orthogonal-matrix-card[of *A*]
by *auto*

lemma *orthogonal-matrix'*: *orthogonal-matrix* (*Q*::*real*^{*n*}^{*n*}) ↔ *Q* ** *transpose*
Q = *mat* 1
by (*metis matrix-left-right-inverse orthogonal-matrix-def*)

lemma *orthogonal-matrix-intro2*:
fixes *A*::*real*^{*n*}^{*n*}
assumes *p*: (*pairwise orthogonal* (rows *A*))
and *n*: ∀ *i*. *norm* (row *i* *A*) = 1
and *c*: *card* (rows *A*) = *nrows* *A*
shows *orthogonal-matrix* *A*
proof (*unfold orthogonal-matrix' vec-eq-iff, clarify, unfold mat-1-fun, auto*)
fix *ia*
have (*A* ** *transpose* *A*) \$ *ia* \$ *ia* = row *ia* *A* · row *ia* *A*
unfolding *matrix-matrix-mult-inner-mult* **unfolding** *column-transpose* ..
also have ... = 1 **using** *n norm-eq-1* **by** *blast*
finally show (*A* ** *transpose* *A*) \$ *ia* \$ *ia* = 1 .
fix *i*
assume *i-not-ia*: *i* ≠ *ia*
have *row-i-not-ia*: row *i* *A* ≠ row *ia* *A*
proof (*rule ccontr, simp*)
assume *row-i-ia*: row *i* *A* = row *ia* *A*
have *rw*: (λ*i*. row *i* *A*)' (*UNIV* - {*ia*}) = {row *i* *A* | *i*. *i* ≠ *ia*} **unfolding** *rows-def*
by *auto*
have *card* (rows *A*) = *card* ({row *i* *A* | *i*. *i* ≠ *ia*})
by (*rule bij-betw-same-card*[of *id*], *unfold bij-betw-def rows-def*)
(*auto, metis row-i-ia i-not-ia*)
also have ... = *card* ((λ*i*. row *i* *A*)' (*UNIV* - {*ia*})) **unfolding** *rw* ..
also have ... ≤ *card* (*UNIV* - {*ia*}) **by** (*metis card-image-le finite-code*)

also have ... $<$ *CARD* ('n) by *simp*
 finally show *False* using *c* unfolding *nrows-def* by *simp*
 qed
 hence *oia*: *orthogonal* (row *i* *A*) (row *ia* *A*)
 using *p* unfolding *pairwise-def* unfolding *rows-def* by *auto*
 have (*A* ** *transpose* *A*) \$ *i* \$ *ia* = row *i* *A* · row *ia* *A*
 unfolding *matrix-matrix-mult-inner-mult* unfolding *column-transpose* ..
 also have ... = 0 using *oia* unfolding *orthogonal-def* .
 finally show (*A* ** *transpose* *A*) \$ *i* \$ *ia* = 0 .
 qed

lemma *is-basis-imp-full-rank*:

fixes *A*::'a::{field} ^'cols::{mod-type} ^'rows::{mod-type}

assumes *b*: *is-basis* (columns *A*)

and *c*: *card* (columns *A*) = *ncols* *A*

shows *rank* *A* = *ncols* *A*

proof –

have *rank* *A* = *col-rank* *A* unfolding *rank-col-rank* ..

also have ... = *vec.dim* (*col-space* *A*) unfolding *col-rank-def* ..

also have ... = *card* (columns *A*)

by (*metis* *b* *col-space-def* *independent-is-basis* *vec.dim-eq-card-independent* *vec.dim-span*)

also have ... = *ncols* *A* using *c* .

finally show ?*thesis* .

qed

lemma *card-columns-le-ncols*:

card (columns *A*) \leq *ncols* *A*

proof –

have *columns-rw*: columns *A* = (λi . column *i* *A*)' *UNIV* unfolding *columns-def*

by *auto*

show ?*thesis* unfolding *columns-rw* *ncols-def* by (*rule* *card-image-le*, *auto*)

qed

lemma *full-rank-imp-is-basis*:

fixes *A*::'a::{field} ^'n::{mod-type} ^'n::{mod-type}

assumes *r*: *rank* *A* = *ncols* *A*

shows *is-basis* (columns *A*) \wedge *card* (columns *A*) = *ncols* *A*

proof (*rule* *conjI*, *unfold* *is-basis-def*, *rule* *conjI*)

have *rank* *A* = *col-rank* *A* unfolding *rank-col-rank* ..

also have ... = *vec.dim* (*col-space* *A*) unfolding *col-rank-def* ..

also have ... = *card* (columns *A*)

by (*metis* (*full-types*) *antisym-conv* *calculation* *card-columns-le-ncols* *col-space-def* *finite-columns* *r* *vec.dim-le-card* *vec.dim-span* *vec.span-superset*)

finally have *: *rank* *A* = *card* (columns *A*) .

then show *c-eq*: *card* (columns *A*) = *ncols* *A* unfolding *r* ..

show *vec.independent* (columns *A*)

by (*metis* * *vec.card-eq-dim-span-indep* *col-rank-def*)

$col\text{-}space\text{-}def$ $finite\text{-}columns$ $rank\text{-}col\text{-}rank$)
thus $vec.span (columns A) = (UNIV::('a ^ n::\{mod\text{-}type\}) set)$
using $independent\text{-}is\text{-}basis[of\ columns\ A]$ $c\text{-}eq$ **unfolding** $is\text{-}basis\text{-}def$ $ncols\text{-}def$
by $simp$
qed

lemma $full\text{-}rank\text{-}imp\text{-}is\text{-}basis2$:
fixes $A::'a::\{field\} ^ n::\{mod\text{-}type\} ^ m::\{mod\text{-}type\}$
assumes $r: rank\ A = ncols\ A$
shows $vec.independent (columns\ A) \wedge vec.span (columns\ A) = col\text{-}space\ A$
 $\wedge card (columns\ A) = ncols\ A$
proof –
have $rank\ A = col\text{-}rank\ A$ **unfolding** $rank\text{-}col\text{-}rank$..
also have $... = vec.dim (col\text{-}space\ A)$ **unfolding** $col\text{-}rank\text{-}def$..
also have $... = card (columns\ A)$
by ($metis (full\text{-}types) antisym\text{-}conv$ $calculation\ card\text{-}columns\text{-}le\text{-}ncols$ $col\text{-}space\text{-}def$
 $finite\text{-}columns$ r $vec.dim\text{-}le\text{-}card$ $vec.dim\text{-}span$ $vec.span\text{-}superset$)
finally have $*$: $rank\ A = card (columns\ A)$.
then have $c\text{-}eq$: $card (columns\ A) = ncols\ A$ **unfolding** r ..
moreover have $vec.independent (columns\ A)$
by ($metis *$ $vec.card\text{-}eq\text{-}dim\text{-}span\text{-}indep$
 $col\text{-}rank\text{-}def$ $col\text{-}space\text{-}def$ $finite\text{-}columns$ $rank\text{-}col\text{-}rank$)
moreover have $vec.span (columns\ A) = col\text{-}space\ A$ **by** ($metis\ col\text{-}space\text{-}def$)
ultimately show $?thesis$ **by** $simp$
qed

corollary $full\text{-}rank\text{-}eq\text{-}is\text{-}basis$:
fixes $A::'a::\{field\} ^ n::\{mod\text{-}type\} ^ n::\{mod\text{-}type\}$
shows $(is\text{-}basis (columns\ A) \wedge (card (columns\ A) = ncols\ A)) = (rank\ A = ncols\ A)$
using $full\text{-}rank\text{-}imp\text{-}is\text{-}basis$ $is\text{-}basis\text{-}imp\text{-}full\text{-}rank$ **by** $blast$

lemma $full\text{-}col\text{-}rank\text{-}imp\text{-}independent\text{-}columns$:
fixes $A::'a::\{field\} ^ n::\{mod\text{-}type\} ^ m::\{mod\text{-}type\}$
assumes $rank\ A = ncols\ A$
shows $vec.independent (columns\ A)$
by ($metis\ assms\ full\text{-}rank\text{-}imp\text{-}is\text{-}basis2$)

lemma $matrix\text{-}vector\text{-}right\text{-}distrib\text{-}minus$:
fixes $A::'a::\{ring\text{-}1\} ^ n ^ m$
shows $A * v (b - c) = (A * v b) - (A * v c)$
proof –
have $A * v (b - c) = A * v (b + - c)$ **by** ($metis\ diff\text{-}minus\text{-}eq\text{-}add$ $minus\text{-}minus$)
also have $... = (A * v b) + (A * v (- c))$ **unfolding** $matrix\text{-}vector\text{-}right\text{-}distrib$..
also have $... = (A * v b) - (A * v c)$
by ($metis (no\text{-}types, opaque\text{-}lifting) add.commute$ $add\text{-}minus\text{-}cancel$
 $matrix\text{-}vector\text{-}right\text{-}distrib$ $uminus\text{-}add\text{-}conv\text{-}diff$)
finally show $?thesis$.

qed

lemma *inv-matrix-vector-mul-left*:

assumes *i*: invertible *A*

shows $(A *v x = A *v y) = (x=y)$

by (*metis i invertible-def matrix-vector-mul-assoc matrix-vector-mul-lid*)

lemma *norm-mult-vec*:

fixes *a*::(real,'b::finite) *vec*

shows $norm (x \cdot x) = norm x * norm x$

by (*metis inner-real-def norm-cauchy-schwarz-eq norm-mult*)

lemma *norm-equivalence*:

fixes *A*::real^{*n*}^{*m*}

shows $((transpose A) *v (A *v x) = 0) \longleftrightarrow (A *v x = 0)$

proof –

have $A *v x = 0$ **if** $transpose A *v (A *v x) = 0$

proof –

have *eq*: $(x v* (transpose A)) = (A *v x)$

by (*metis transpose-transpose transpose-vector*)

have *eq-0*: $0 = (x v* (transpose A)) * (A *v x)$

by *auto* (*metis that dot-lmul-matrix inner-eq-zero-iff inner-zero-left mult-not-zero transpose-vector*)

hence $0 = norm ((x v* (transpose A)) * (A *v x))$ **by** *auto*

also have $... = norm ((A *v x)*(A *v x))$ **unfolding** *eq* ..

also have $... = norm ((A *v x) \cdot (A *v x))$

by (*metis eq-0 that dot-lmul-matrix eq inner-zero-right norm-zero*)

also have $... = norm (A *v x)^2$ **unfolding** *norm-mult-vec*[*of (A *v x)*]

power2-eq-square ..

finally show $A *v x = 0$

by *simp*

qed

then show *?thesis*

by *auto*

qed

lemma *invertible-transpose-mult*:

fixes *A*::real^{*cols*}::{*mod-type*}^{*rows*}::{*mod-type*}

assumes *r*: rank *A* = *ncols A*

shows invertible ($transpose A ** A$)

proof –

have *null-eq*: null-space *A* = null-space ($transpose A ** A$)

proof (*auto*)

fix *x* **assume** *x*: *x* ∈ null-space *A*

show *x* ∈ null-space ($transpose A ** A$) **using** *x* **unfolding** *null-space-def*

by (*metis (lifting, full-types) matrix-vector-mul-assoc matrix-vector-mult-0-right mem-Collect-eq*)

next

```

fix  $x$  assume  $x: x \in \text{null-space } (\text{transpose } A ** A)$ 
show  $x \in \text{null-space } A$ 
  by (metis is-solution-def matrix-vector-mul-assoc mem-Collect-eq
    norm-equivalence null-space-eq-solution-set solution-set-def x)
qed
have  $\text{rank } A = \text{vec.dim } (\text{UNIV}::(\text{real}^{\wedge}\text{cols}::\{\text{mod-type}\}) \text{ set}) - \text{vec.dim } (\text{null-space } A)$ 
  using rank-nullity-theorem-matrices[of  $A$ ]
  unfolding rank-eq-dim-col-space'[of  $A$ , symmetric]
  by (simp only: add commute diff-add-inverse2 ncols-def vec-dim-card)
also have  $\dots = \text{vec.dim } (\text{UNIV}::(\text{real}^{\wedge}\text{cols}::\{\text{mod-type}\}) \text{ set}) - \text{vec.dim } (\text{null-space } (\text{transpose } A ** A))$ 
  unfolding null-eq ..
also have  $\dots = \text{rank } (\text{transpose } A ** A)$ 
  by (metis add commute diff-add-inverse2 ncols-def rank-eq-dim-col-space
    rank-nullity-theorem-matrices vec-dim-card)
finally have  $r\text{-}A: \text{rank } A = \text{rank } (\text{transpose } A ** A)$  .
show ?thesis using full-rank-implies-invertible r unfolding ncols-def nrows-def r-A .
qed

```

```

lemma matrix-inv-mult:
  fixes  $A::'a::\{\text{semiring-1}\}^{\wedge}n^{\wedge}n$ 
  and  $B::'a::\{\text{semiring-1}\}^{\wedge}n^{\wedge}n$ 
  assumes invertible A and invertible B
  shows  $\text{matrix-inv } (A ** B) = \text{matrix-inv } B ** \text{matrix-inv } A$ 
proof (rule matrix-inv-unique[of  $A**B$ ])
  show  $A ** B ** (\text{matrix-inv } B ** \text{matrix-inv } A) = \text{mat } 1$ 
    by (metis assms(1) assms(2) matrix-inv-right matrix-mul-assoc matrix-mul-lid)
  show  $\text{matrix-inv } B ** \text{matrix-inv } A ** (A ** B) = \text{mat } 1$ 
    by (metis assms(1) assms(2) matrix-inv-left matrix-mul-assoc matrix-mul-lid)
qed

```

```

lemma invertible-transpose:
  fixes  $A::'a::\{\text{field}\}^{\wedge}n^{\wedge}n$ 
  assumes invertible A
  shows invertible (transpose A)
  by (metis invertible-det-nz assms det-transpose)

```

The following lemmas are generalizations of some parts of the library. They should be in the file *Generalizations.thy* of the Gauss-Jordan AFP entry.

```

context vector-space
begin
lemma span-eq:  $(\text{span } S = \text{span } T) = (S \subseteq \text{span } T \wedge T \subseteq \text{span } S)$ 
  using span-superset[unfolded subset-eq] using span-mono[of  $T \text{ span } S$ ] span-mono[of  $S \text{ span } T$ ]
  by (auto simp add: span-span)
end

```

```

lemma basis-orthogonal:
  fixes  $B :: 'a::\text{real-inner set}$ 
  assumes  $fB$ : finite B
  shows  $\exists C. \text{finite } C \wedge \text{card } C \leq \text{card } B \wedge \text{span } C$ 
     $= \text{span } B \wedge \text{pairwise orthogonal } C$ 
  (is  $\exists C. ?P B C$ )
  using  $fB$ 
proof (induct rule: finite-induct)
  case empty
  then show  $?case$ 
    apply (rule exI[where x={}])
    apply (auto simp add: pairwise-def)
    done
next
  case (insert a B)
  note  $fB = \langle \text{finite } B \rangle$  and  $aB = \langle a \notin B \rangle$ 
  from  $\langle \exists C. \text{finite } C \wedge \text{card } C \leq \text{card } B \wedge \text{span } C = \text{span } B \wedge \text{pairwise orthogonal } C \rangle$ 
  obtain  $C$  where  $C$ : finite C card C ≤ card B
     $\text{span } C = \text{span } B$  pairwise orthogonal C by blast
  let  $?a = a - \text{sum } (\lambda x. (x \cdot a / (x \cdot x)) *_{\mathbb{R}} x) C$ 
  let  $?C = \text{insert } ?a C$ 
  from  $C(1)$  have  $fC$ : finite ?C
    by simp
  from  $fB aB C(1,2)$  have  $cC$ :  $\text{card } ?C \leq \text{card } (\text{insert } a B)$ 
    by (simp add: card-insert-if)
  {
    fix  $x k$ 
    have  $th0$ :  $\bigwedge (a::'a) b c. a - (b - c) = c + (a - b)$ 
      by (simp add: field-simps)
    have  $x - k *_{\mathbb{R}} (a - (\sum_{x \in C}. (x \cdot a / (x \cdot x)) *_{\mathbb{R}} x)) \in \text{span } C$ 
       $\longleftrightarrow x - k *_{\mathbb{R}} a \in \text{span } C$ 
      apply (simp only: scaleR-right-diff-distrib th0)
      apply (rule span-add-eq)
      apply (rule span-mul)
      apply (rule span-sum)
      apply (rule span-mul)
      apply (rule span-base)
      apply assumption
      done
  }
  then have  $SC$ :  $\text{span } ?C = \text{span } (\text{insert } a B)$ 
    unfolding set-eq-iff span-breakdown-eq C(3)[symmetric] by auto
  {
    fix  $y$ 
    assume  $yC$ :  $y \in C$ 
    then have  $Cy$ :  $C = \text{insert } y (C - \{y\})$ 
      by blast
  }

```

```

have fth: finite (C - {y})
  using C by simp
have orthogonal ?a y
  unfolding orthogonal-def
  unfolding inner-diff inner-sum-left right-minus-eq
  unfolding sum.remove [OF ⟨finite C⟩ ⟨y ∈ C⟩]
  apply (clarsimp simp add: inner-commute[of y a])
  apply (rule sum.neutral)
  apply carsimp
  apply (rule C(4)[unfolded pairwise-def orthogonal-def, rule-format])
  using ⟨y ∈ C⟩ by auto
}
with ⟨pairwise orthogonal C⟩ have CPO: pairwise orthogonal ?C
  by (rule pairwise-orthogonal-insert)
from fC cC SC CPO have ?P (insert a B) ?C
  by blast
then show ?case by blast
qed

```

```

lemma op-vec-scaleR: (*s) = (*R)
  by (force simp: scalar-mult-eq-scaleR)

```

end

2 Projections

```

theory Projections
imports
  Miscellaneous-QR
begin

```

2.1 Definitions of vector projection and projection of a vector onto a set.

```

definition proj v u = (v · u / (u · u)) *R u

```

```

definition proj-onto a S = (sum (λx. proj a x) S)

```

2.2 Properties

```

lemma proj-onto-sum-rw:
  sum (λx. (x · v / (x · x)) *R x) A = sum (λx. (v · x / (x · x)) *R x) A
  by (rule sum.cong, auto simp add: inner-commute)

```

```

lemma vector-sub-project-orthogonal-proj:
  fixes b x :: 'a::euclidean-space
  shows inner b (x - proj x b) = 0
  using vector-sub-project-orthogonal
  unfolding proj-def inner-commute[of x b]

```

by *auto*

lemma *orthogonal-proj-set*:

assumes yC : $y \in C$ **and** C : *finite* C **and** p : *pairwise orthogonal* C

shows *orthogonal* ($a - \text{proj-onto } a \ C$) y

proof –

have Cy : $C = \text{insert } y \ (C - \{y\})$ **using** yC

by *blast*

have fth : *finite* ($C - \{y\}$)

using C **by** *simp*

show *orthogonal* ($a - \text{proj-onto } a \ C$) y

unfolding *orthogonal-def* **unfolding** *proj-onto-def* **unfolding** *proj-def*[*abs-def*]

unfolding *inner-diff*

unfolding *inner-sum-left*

unfolding *right-minus-eq*

unfolding *sum.remove*[*OF* $C \ yC$]

apply (*clarsimp simp add: inner-commute*[*of* $y \ a$])

apply (*rule sum.neutral*)

apply *clarsimp*

apply (*rule p*[*unfolded pairwise-def orthogonal-def, rule-format*])

using yC **by** *auto*

qed

lemma *pairwise-orthogonal-proj-set*:

assumes C : *finite* C **and** p : *pairwise orthogonal* C

shows *pairwise orthogonal* ($\text{insert } (a - \text{proj-onto } a \ C) \ C$)

by (*rule pairwise-orthogonal-insert*[*OF* p], *auto simp add: orthogonal-proj-set* C p)

2.3 Orthogonal Complement

definition *orthogonal-complement* $W = \{x. \forall y \in W. \text{orthogonal } x \ y\}$

lemma *in-orthogonal-complement-imp-orthogonal*:

assumes x : $y \in S$

and $x \in \text{orthogonal-complement } S$

shows *orthogonal* $x \ y$

using *assms orthogonal-commute*

unfolding *orthogonal-complement-def*

by *blast*

lemma *subspace-orthogonal-complement*: *subspace* (*orthogonal-complement* W)

unfolding *subspace-def orthogonal-complement-def*

by (*simp add: orthogonal-def inner-left-distrib*)

lemma *orthogonal-complement-mono*:

assumes A -*in*- B : $A \subseteq B$

shows *orthogonal-complement* $B \subseteq \text{orthogonal-complement } A$

proof
fix x **assume** $x: x \in \text{orthogonal-complement } B$
show $x \in \text{orthogonal-complement } A$ **using** x **unfolding** *orthogonal-complement-def*
by (*simp add: orthogonal-def, metis A-in-B in-mono*)
qed

lemma *B-in-orthogonal-complement-of-orthogonal-complement:*
shows $B \subseteq \text{orthogonal-complement } (\text{orthogonal-complement } B)$
by (*auto simp add: orthogonal-complement-def orthogonal-def inner-commute*)

lemma *pythagorean-theorem-norm:*
assumes $o: \text{orthogonal } x \ y$
shows $\text{norm } (x+y)^{\wedge 2} = \text{norm } x^{\wedge 2} + \text{norm } y^{\wedge 2}$
proof –
have $\text{norm } (x+y)^{\wedge 2} = (x+y) \cdot (x+y)$ **unfolding** *power2-norm-eq-inner* ..
also have $\dots = ((x+y) \cdot x) + ((x+y) \cdot y)$ **unfolding** *inner-right-distrib* ..
also have $\dots = (x \cdot x) + (x \cdot y) + (y \cdot x) + (y \cdot y)$
unfolding *real-inner-class.inner-add-left* **by** *simp*
also have $\dots = (x \cdot x) + (y \cdot y)$ **using** o **unfolding** *orthogonal-def*
by (*metis monoid-add-class.add.right-neutral inner-commute*)
also have $\dots = \text{norm } x^{\wedge 2} + \text{norm } y^{\wedge 2}$ **unfolding** *power2-norm-eq-inner* ..
finally show *?thesis* .
qed

lemma *in-orthogonal-complement-basis:*
fixes $B::'a::\{\text{euclidean-space}\}$ *set*
assumes $S: \text{subspace } S$
and $\text{ind-}B: \text{independent } B$
and $B: B \subseteq S$
and $\text{span-}B: S \subseteq \text{span } B$
shows $(v \in \text{orthogonal-complement } S) = (\forall a \in B. \text{orthogonal } a \ v)$
proof (*unfold orthogonal-complement-def, auto*)
fix a **assume** $\forall x \in S. \text{orthogonal } v \ x$ **and** $a \in B$
thus $\text{orthogonal } a \ v$
by (*metis B orthogonal-commute rev-subsetD*)
next
fix x **assume** $o: \forall a \in B. \text{orthogonal } a \ v$ **and** $x: x \in S$
have $\text{finite-}B: \text{finite } B$ **using** *independent-bound-general[OF ind-B]* ..
have $\text{span-}B\text{-eq}: S = \text{span } B$ **using** $B \ S \ \text{span-}B \ \text{span-subspace}$ **by** *blast*
obtain f **where** $f: (\sum a \in B. f \ a \ *_R \ a) = x$ **using** *span-finite[OF finite-B]*
using x **unfolding** *span-B-eq* **by** *force*
have $v \cdot x = v \cdot (\sum a \in B. f \ a \ *_R \ a)$ **unfolding** f ..
also have $\dots = (\sum a \in B. v \cdot (f \ a \ *_R \ a))$ **unfolding** *inner-sum-right* ..
also have $\dots = (\sum a \in B. f \ a \ * \ (v \cdot a))$ **unfolding** *inner-scaleR-right* ..
also have $\dots = 0$ **using** *sum.neutral o* **by** (*simp add: orthogonal-def inner-commute*)
finally show $\text{orthogonal } v \ x$ **unfolding** *orthogonal-def* .
qed

See https://people.math.osu.edu/husen.1/teaching/571/least_squares.pdf

Part 1 of the Theorem 1.7 in the previous website, but the proof has been carried out in other way.

lemma *v-minus-p-orthogonal-complement*:
fixes $X::'a::\{euclidean-space\}$ *set*
assumes *subspace-S: subspace S*
and *ind-X: independent X*
and $X: X \subseteq S$
and *span-X: $S \subseteq \text{span } X$*
and *o: pairwise orthogonal X*
shows $(v - \text{proj-onto } v X) \in \text{orthogonal-complement } S$
unfolding *in-orthogonal-complement-basis[OF subspace-S ind-X X span-X]*
proof
fix a **assume** $a: a \in X$
let $?p = \text{proj-onto } v X$
show *orthogonal a (v - ?p)*
unfolding *orthogonal-commute[of a v-?p]*
by (*rule orthogonal-proj-set[OF a - o]*)
(simp add: independent-bound-general[OF ind-X])
qed

Part 2 of the Theorem 1.7 in the previous website.

lemma *UNIV-orthogonal-complement-decomposition*:
fixes $S::'a::\{euclidean-space\}$ *set*
assumes *s: subspace S*
shows $UNIV = S + (\text{orthogonal-complement } S)$
proof (*unfold set-plus-def, auto*)
fix v
obtain X **where** *ind-X: independent X*
and $X: X \subseteq S$
and *span-X: $S \subseteq \text{span } X$*
and *o: pairwise orthogonal X*
by (*metis order-refl orthonormal-basis-subspace s*)
have *finite-X: finite X* **by** (*metis independent-bound-general ind-X*)
let $?p = \text{proj-onto } v X$
have $v = ?p + (v - ?p)$ **by** *simp*
moreover **have** $?p \in S$ **unfolding** *proj-onto-def proj-def[abs-def]*
by (*rule subspace-sum[OF s]*)
(simp add: X s rev-subsetD subspace-mul)
moreover **have** $(v - ?p) \in \text{orthogonal-complement } S$
by (*rule v-minus-p-orthogonal-complement[OF s ind-X X span-X o]*)
ultimately show $\exists a \in S. \exists b \in \text{orthogonal-complement } S. v = a + b$ **by** *force*
qed

2.4 Normalization of vectors

definition *normalize*
where $\text{normalize } x = ((1/\text{norm } x) *_{\mathbb{R}} x)$
definition *normalize-set-of-vec*
where $\text{normalize-set-of-vec } X = \text{normalize}' X$

```

lemma norm-normalize:
  assumes  $x \neq 0$ 
  shows  $\text{norm} (\text{normalize } x) = 1$ 
  by (simp add: normalize-def assms)

lemma normalize-0:  $(\text{normalize } x = 0) = (x = 0)$ 
  unfolding normalize-def by auto

lemma norm-normalize-set-of-vec:
  assumes  $x \neq 0$ 
  and  $x \in \text{normalize-set-of-vec } X$ 
  shows  $\text{norm } x = 1$ 
  using assms norm-normalize normalize-0 unfolding normalize-set-of-vec-def by
blast

end

```

3 The Gram-Schmidt algorithm

```

theory Gram-Schmidt
imports
  Miscellaneous-QR
  Projections
begin

```

3.1 Gram-Schmidt algorithm

The algorithm is used to orthogonalise a set of vectors. The Gram-Schmidt process takes a set of vectors S and generates another orthogonal set that spans the same subspace as S .

We present three ways to compute the Gram-Schmidt algorithm.

1. The first one has been developed thinking about the simplicity of its formalisation. Given a list of vectors, the output is another list of orthogonal vectors with the same span. Such a list is constructed following the Gram-Schmidt process presented in any book, but in the reverse order (starting the process from the last element of the input list).
2. Based on previous formalization, another function has been defined to compute the process of the Gram-Schmidt algorithm in the natural order (starting from the first element of the input list).
3. The third way has as input and output a matrix. The algorithm is applied to the columns of a matrix, obtaining a matrix whose columns

are orthogonal and where the column space is kept. This will be a previous step to compute the QR decomposition.

Every function can be executed with arbitrary precision (using rational numbers).

3.1.1 First way

definition *Gram-Schmidt-step* :: ('a::{real-inner} ^b) => ('a ^b) list => ('a ^b) list

where *Gram-Schmidt-step* a ys = ys @ [(a - proj-onto a (set ys))]

definition *Gram-Schmidt* xs = foldr *Gram-Schmidt-step* xs []

lemma *Gram-Schmidt-cons*:

Gram-Schmidt (a # xs) = *Gram-Schmidt-step* a (*Gram-Schmidt* xs)

unfolding *Gram-Schmidt-def* **by** *auto*

lemma *basis-orthogonal'*:

fixes xs::('a::{real-inner} ^b) list

shows *length* (*Gram-Schmidt* xs) = *length* (xs) ∧
span (set (*Gram-Schmidt* xs)) = *span* (set xs) ∧
pairwise orthogonal (set (*Gram-Schmidt* xs))

proof (*induct* xs)

case *Nil*

show ?case **unfolding** *Gram-Schmidt-def* *pairwise-def* **by** *fastforce*

next

case (*Cons* a xs)

have l: *length* (*Gram-Schmidt* xs) = *length* xs

and s: *span* (set (*Gram-Schmidt* xs)) = *span* (set xs)

and p: *pairwise orthogonal* (set (*Gram-Schmidt* xs)) **using** *Cons.hyps* **by** *auto*

show *length* (*Gram-Schmidt* (a # xs)) = *length* (a # xs) ∧
span (set (*Gram-Schmidt* (a # xs))) = *span* (set (a # xs))
∧ *pairwise orthogonal* (set (*Gram-Schmidt* (a # xs)))

proof

show *length* (*Gram-Schmidt* (a # xs)) = *length* (a # xs)

unfolding *Gram-Schmidt-cons* **unfolding** *Gram-Schmidt-step-def* **using** l **by**

auto

show *span* (set (*Gram-Schmidt* (a # xs)))

= *span* (set (a # xs)) ∧ *pairwise orthogonal* (set (*Gram-Schmidt* (a # xs)))

proof

have *set-rw1*: set (a # xs) = insert a (set xs) **by** *simp*

have *set-rw2*: set (*Gram-Schmidt* (a # xs))

= (insert (a - (∑ x∈set (*Gram-Schmidt* xs). (a · x / (x · x)) *_R x)) (set (*Gram-Schmidt* xs)))

unfolding *Gram-Schmidt-cons* *Gram-Schmidt-step-def* *proj-onto-def* *proj-def*[*abs-def*]

by *auto*

define C **where** C = set (*Gram-Schmidt* xs)

have *finite-C*: *finite* C **unfolding** *C-def* **by** *auto*

```

{
  fix x k
  have th0: !(a::'a^b) b c. a - (b - c) = c + (a - b)
    by (simp add: field-simps)
  have x - k *R (a - (∑x∈C. (a · x / (x · x)) *R x)) ∈ span C
    ←→ x - k *R a ∈ span C
    apply (simp only: scaleR-right-diff-distrib th0)
    apply (rule span-add-eq)
    apply (rule span-mul)
    apply (rule span-sum)
    apply (rule span-mul)
    apply (rule span-base)
    apply assumption
  done
}
then show span (set (Gram-Schmidt (a # xs))) = span (set (a # xs))
  unfolding set-eq-iff set-rw2 set-rw1 span-breakdown-eq C-def s[symmetric]
  by auto
with p show pairwise orthogonal (set (Gram-Schmidt (a # xs)))
  using pairwise-orthogonal-proj-set[OF finite-C]
  unfolding set-rw2 unfolding C-def proj-def[symmetric] proj-onto-def[symmetric]
by simp
  qed
  qed
qed

```

```

lemma card-Gram-Schmidt:
  fixes xs::('a::{real-inner})^b list
  assumes distinct xs
  shows card(set (Gram-Schmidt xs)) ≤ card (set (xs))
  using assms
proof (induct xs)
  case Nil show ?case unfolding Gram-Schmidt-def by simp
next
  case (Cons x xs)
  define b where b = (∑xa∈set (Gram-Schmidt xs). (x · xa / (xa · xa)) *R xa)
  have distinct-xs: distinct xs using Cons.prem by auto
  show ?case
  proof (cases x - b ∉ set (Gram-Schmidt xs))
    case True
  have card (set (Gram-Schmidt (x # xs))) = card (insert (x - b) (set (Gram-Schmidt xs)))
    unfolding Gram-Schmidt-cons Gram-Schmidt-step-def b-def
    unfolding proj-onto-def proj-def[abs-def] by simp
  also have ... = Suc (card (set (Gram-Schmidt xs)))
  proof (rule card-insert-disjoint)
    show finite (set (Gram-Schmidt xs)) by simp
    show x - b ∉ set (Gram-Schmidt xs) using True .
  
```

```

qed
also have ... ≤ Suc (card (set xs)) using Cons.hyps[OF distinct-xs] by simp
also have ... = Suc (length (remdups xs)) unfolding List.card-set ..
also have ... ≤ (length (remdups (x # xs)))
by (metis Cons.prem1 distinct-xs impossible-Cons not-less-eq-eq remdups-id-iff-distinct)
also have ... ≤ (card (set (x # xs)))
by (metis dual-order.refl length-remdups-card-conv)
finally show ?thesis .
next
case False
have x-b-in: x - b ∈ set (Gram-Schmidt xs) using False by simp
have card (set (Gram-Schmidt (x # xs))) = card (insert (x - b) (set (Gram-Schmidt
xs)))
unfolding Gram-Schmidt-cons Gram-Schmidt-step-def b-def
unfolding proj-onto-def proj-def[abs-def] by simp
also have ... = (card (set (Gram-Schmidt xs))) by (metis False insert-absorb)
also have ... ≤ (card (set xs)) using Cons.hyps[OF distinct-xs] .
also have ... ≤ (card (set (x # xs))) unfolding List.card-set by simp
finally show ?thesis .
qed
qed

```

lemma *orthogonal-basis-exists:*

```

fixes V :: (real^b) list
assumes B: is-basis (set V)
and d: distinct V
shows vec.independent (set (Gram-Schmidt V)) ∧ (set V) ⊆ vec.span (set
(Gram-Schmidt V))
∧ (card (set (Gram-Schmidt V)) = vec.dim (set V)) ∧ pairwise orthogonal (set
(Gram-Schmidt V))
proof -
have (set V) ⊆ vec.span (set (Gram-Schmidt V))
using basis-orthogonal'[of V]
using vec.span-superset[where ?'a=real, where ?'b='b]
by (auto simp: span-vec-eq)
moreover have pairwise orthogonal (set (Gram-Schmidt V))
using basis-orthogonal'[of V] by blast
moreover have c: (card (set (Gram-Schmidt V)) = vec.dim (set V))
proof -
have card-eq-dim: card (set V) = vec.dim (set V)
by (metis B independent-is-basis vec.dim-span vec.indep-card-eq-dim-span)
have vec.dim (set V) ≤ (card (set (Gram-Schmidt V))) using B unfolding
is-basis-def
using vec.independent-span-bound[of (set (Gram-Schmidt V)) set V]
using basis-orthogonal'[of V]
by (simp add: calculation(1) local.card-eq-dim)
moreover have (card (set (Gram-Schmidt V))) ≤ vec.dim (set V)
using card-Gram-Schmidt[OF d] card-eq-dim by auto
ultimately show ?thesis by auto

```

qed
moreover have $\text{vec.independent (set (Gram-Schmidt V))}$
proof (rule $\text{vec.card-le-dim-spanning[of - UNIV::(real^b) set]}$)
show $\text{set (Gram-Schmidt V) } \subseteq \text{(UNIV::(real^b) set)}$ **by simp**
show $\text{UNIV } \subseteq \text{vec.span (set (Gram-Schmidt V))}$
using $\text{basis-orthogonal'[of V]}$ **using** B **unfolding** is-basis-def
by (simp add: span-vec-eq)
show $\text{finite (set (Gram-Schmidt V))}$ **by simp**
show $\text{card (set (Gram-Schmidt V)) } \leq \text{vec.dim (UNIV::(real^b) set)}$
by (metis $\text{c top-greatest vec.dim-subset}$)
qed
ultimately show $?thesis$ **by simp**
qed

corollary $\text{orthogonal-basis-exists'}$:
fixes $V :: (\text{real}^b) \text{ list}$
assumes B : is-basis (set V)
and d : distinct V
shows $\text{is-basis (set (Gram-Schmidt V))}$
 $\wedge \text{distinct (Gram-Schmidt V) } \wedge \text{pairwise orthogonal (set (Gram-Schmidt V))}$
using B $\text{orthogonal-basis-exists}$ basis-orthogonal' $\text{card-distinct } d$
 vec.dim-unique distinct-card is-basis-def subset-refl
by (metis span-vec-eq)

3.1.2 Second way

This definition applies the Gram Schmidt process starting from the first element of the list.

definition $\text{Gram-Schmidt2 } xs = \text{Gram-Schmidt (rev xs)}$

lemma basis-orthogonal2 :
fixes $xs :: ('a :: \{\text{real-inner}\}^b) \text{ list}$
shows $\text{length (Gram-Schmidt2 } xs) = \text{length (xs)}$
 $\wedge \text{span (set (Gram-Schmidt2 } xs)) = \text{span (set xs)}$
 $\wedge \text{pairwise orthogonal (set (Gram-Schmidt2 } xs))}$
by (metis Gram-Schmidt2-def basis-orthogonal' length-rev set-rev)

lemma $\text{card-Gram-Schmidt2}$:
fixes $xs :: ('a :: \{\text{real-inner}\}^b) \text{ list}$
assumes $\text{distinct } xs$
shows $\text{card (set (Gram-Schmidt2 } xs)) } \leq \text{card (set (xs))}$
by (metis Gram-Schmidt2-def assms card-Gram-Schmidt distinct-rev set-rev)

lemma $\text{orthogonal-basis-exists2}$:
fixes $V :: (\text{real}^b) \text{ list}$
assumes B : is-basis (set V)
and d : distinct V
shows $\text{vec.independent (set (Gram-Schmidt2 V)) } \wedge \text{(set V) } \subseteq \text{vec.span (set$

(Gram-Schmidt2 V)
 \wedge (card (set (Gram-Schmidt2 V)) = vec.dim (set V)) \wedge pairwise orthogonal (set (Gram-Schmidt2 V))
by (metis Gram-Schmidt.orthogonal-basis-exists Gram-Schmidt2-def distinct-rev set-rev B basis-orthogonal2 d)

3.1.3 Third way

The following definitions applies the Gram Schmidt process in the columns of a given matrix. It is previous step to the computation of the QR decomposition.

definition Gram-Schmidt-column-k :: 'a::{real-inner} ^ cols::{mod-type} ^ rows \Rightarrow nat
 \Rightarrow 'a ^ cols::{mod-type} ^ rows
where Gram-Schmidt-column-k A k
 = (χ a. (χ b. (if b = from-nat k then (column b A - (proj-onto (column b A) {column i A | i. i < b})) else (column b A)) \$ a))

definition Gram-Schmidt-upt-k A k = foldl Gram-Schmidt-column-k A [0..<(Suc k)]

definition Gram-Schmidt-matrix A = Gram-Schmidt-upt-k A (ncols A - 1)

Some definitions and lemmas in order to get execution.

definition Gram-Schmidt-column-k-row A k a =
 vec-lambda(λ b. (if b = from-nat k then (column b A - (\sum x \in {column i A | i. i < b}. ((column b A) \cdot x / (x \cdot x)) *_R x)) else (column b A)) \$ a)

lemma Gram-Schmidt-column-k-row-code[code abstract]:
 vec-nth (Gram-Schmidt-column-k-row A k a)
 = (%b. (if b = from-nat k then (column b A - (\sum x \in {column i A | i. i < b}. ((column b A) \cdot x / (x \cdot x)) *_R x)) else (column b A)) \$ a)
unfolding Gram-Schmidt-column-k-row-def
by (metis (lifting) vec-lambda-beta)

lemma Gram-Schmidt-column-k-code[code abstract]:
 vec-nth (Gram-Schmidt-column-k A k) = Gram-Schmidt-column-k-row A k
unfolding Gram-Schmidt-column-k-def **unfolding** Gram-Schmidt-column-k-row-def[abs-def]
unfolding proj-onto-def proj-def[abs-def]
by fastforce

Proofs

lemma Gram-Schmidt-upt-k-suc:

$\text{Gram-Schmidt-upt-}k\ A\ (\text{Suc } k) = (\text{Gram-Schmidt-column-}k\ (\text{Gram-Schmidt-upt-}k\ A\ k)\ (\text{Suc } k))$

proof –

have $rw: [0..<\text{Suc } (\text{Suc } k)] = [0..<\text{Suc } k] @ [(\text{Suc } k)]$ **by** *simp*

show *?thesis unfolding Gram-Schmidt-upt-k-def*

unfolding *rw unfolding foldl-append unfolding foldl.simps ..*

qed

lemma *column-Gram-Schmidt-upt-k-preserves:*

fixes $A::'a::\{\text{real-inner}\}^{\sim}\text{cols}::\{\text{mod-type}\}^{\sim}\text{rows}$

assumes $i\text{-less-suc}: \text{to-nat } i < (\text{Suc } k)$

and $\text{suc-less-card}: \text{Suc } k < \text{CARD } ('cols)$

shows $\text{column } i\ (\text{Gram-Schmidt-upt-}k\ A\ (\text{Suc } k)) = \text{column } i\ (\text{Gram-Schmidt-upt-}k\ A\ k)$

proof –

have $\text{column } i\ (\text{Gram-Schmidt-upt-}k\ A\ (\text{Suc } k))$

$= \text{column } i\ (\text{Gram-Schmidt-column-}k\ (\text{Gram-Schmidt-upt-}k\ A\ k)\ (\text{Suc } k))$

unfolding *Gram-Schmidt-upt-k-suc ..*

also have $\dots = \text{column } i\ (\text{Gram-Schmidt-upt-}k\ A\ k)$ **unfolding** *Gram-Schmidt-column-k-def column-def using i-less-suc by (auto simp add: to-nat-from-nat-id[OF suc-less-card])*

finally show *?thesis .*

qed

lemma *column-set-Gram-Schmidt-upt-k:*

fixes $A::'a::\{\text{real-inner}\}^{\sim}\text{cols}::\{\text{mod-type}\}^{\sim}\text{rows}$

assumes $k: \text{Suc } k < \text{CARD } ('cols)$

shows $\{\text{column } i\ (\text{Gram-Schmidt-upt-}k\ A\ (\text{Suc } k)) \mid i. \text{to-nat } i \leq (\text{Suc } k)\} =$
 $\{\text{column } i\ (\text{Gram-Schmidt-upt-}k\ A\ k) \mid i. \text{to-nat } i \leq k\} \cup \{\text{column } (\text{from-nat } (\text{Suc } k))\ (\text{Gram-Schmidt-upt-}k\ A\ k)$

$- (\sum x \in \{\text{column } i\ (\text{Gram-Schmidt-upt-}k\ A\ k) \mid i. \text{to-nat } i \leq k\}. (x \cdot (\text{column } (\text{from-nat } (\text{Suc } k))\ (\text{Gram-Schmidt-upt-}k\ A\ k)) / (x \cdot x)) *_R x)\}$

proof –

have $\text{set-rw}: \{\chi\ ia. \text{Gram-Schmidt-upt-}k\ A\ k\ \$\ ia\ \$\ i\ \mid i.$

$i < \text{from-nat } (\text{Suc } k)\} = \{\chi\ ia. \text{Gram-Schmidt-upt-}k\ A\ k\ \$\ ia\ \$\ i\ \mid i. \text{to-nat } i \leq k\}$

proof *(auto, vector, metis less-Suc-eq-le to-nat-le)*

fix $i::'cols$

assume $\text{to-nat } i \leq k$

hence $\text{to-nat } i < \text{Suc } k$ **by** *simp*

hence $i\text{-less-suc}: i < \text{from-nat } (\text{Suc } k)$ **using** *from-nat-le[OF - k]* **by** *simp*

show $\exists l. (\lambda j. \text{Gram-Schmidt-upt-}k\ A\ k\ \$\ j\ \$\ i) = (\lambda j'. \text{Gram-Schmidt-upt-}k\ A\ k\ \$\ j'\ \$\ l) \wedge l < \text{mod-type-class.from-nat } (\text{Suc } k)$

by *(rule exI[of - i], simp add: i-less-suc)*

qed

have $rw: [0..<\text{Suc } (\text{Suc } k)] = [0..<\text{Suc } k] @ [(\text{Suc } k)]$ **by** *simp*

have $\{\text{column } i\ (\text{Gram-Schmidt-upt-}k\ A\ (\text{Suc } k)) \mid i. \text{to-nat } i \leq (\text{Suc } k)\}$

$= \{\text{column } i\ (\text{Gram-Schmidt-column-}k\ (\text{Gram-Schmidt-upt-}k\ A\ k)\ (\text{Suc } k)) \mid i.$

$\text{to-nat } i \leq \text{Suc } k\}$

unfolding *Gram-Schmidt-upt-k-def*

unfolding *rw* **unfolding** *foldl-append* **unfolding** *foldl.simps ..*
also have ... = {*column i (Gram-Schmidt-upt-k A k) | i. to-nat i ≤ k*} ∪ {*column (from-nat (Suc k)) (Gram-Schmidt-upt-k A k)*
– ($\sum_{x \in \{\text{column } i \text{ (Gram-Schmidt-upt-k A k) | i. to-nat } i \leq k\}} (x \cdot (\text{column (from-nat (Suc k)) (Gram-Schmidt-upt-k A k)) / (x \cdot x)) *_R x)$)}
proof (*auto*)
fix *i::'cols*
assume *ik: to-nat i ≤ k*
show $\exists ia. \text{column } i \text{ (Gram-Schmidt-upt-k A k)}$
= *column ia (Gram-Schmidt-column-k (Gram-Schmidt-upt-k A k) (Suc k))* ∧
to-nat ia ≤ Suc k
proof (*rule exI[of - i], rule conjI*)
have *i-less-suc: to-nat i < Suc k* **using** *ik* **by** *auto*
thus *to-nat i ≤ Suc k* **by** *simp*
show *column i (Gram-Schmidt-upt-k A k) = column i (Gram-Schmidt-column-k (Gram-Schmidt-upt-k A k) (Suc k))*
using *column-Gram-Schmidt-upt-k-preserves[OF i-less-suc k, of A]*
unfolding *Gram-Schmidt-upt-k-suc ..*
qed
next
show $\exists a. \text{column (from-nat (Suc k)) (Gram-Schmidt-upt-k A k) -}$
($\sum_{x \in \{\text{column } i \text{ (Gram-Schmidt-upt-k A k) | i. to-nat } i \leq k\}} (x \cdot \text{column (from-nat (Suc k)) (Gram-Schmidt-upt-k A k) / (x \cdot x)) *_R x)$) =
column a (Gram-Schmidt-column-k (Gram-Schmidt-upt-k A k) (Suc k)) ∧
to-nat a ≤ Suc k
proof (*rule exI[of - from-nat (Suc k)::'cols], rule conjI*)

show *to-nat (from-nat (Suc k)::'cols) ≤ Suc k* **unfolding** *to-nat-from-nat-id[OF k] ..*
show *column (from-nat (Suc k)) (Gram-Schmidt-upt-k A k) -*
($\sum_{x \in \{\text{column } i \text{ (Gram-Schmidt-upt-k A k) | i. to-nat } i \leq k\}} (x \cdot \text{column (from-nat (Suc k)) (Gram-Schmidt-upt-k A k) / (x \cdot x)) *_R x)$) =
column (from-nat (Suc k)) (Gram-Schmidt-column-k (Gram-Schmidt-upt-k A k) (Suc k))
unfolding *Gram-Schmidt-column-k-def column-def* **apply** *auto* **unfolding**
set-rw
unfolding *vector-scaleR-component[symmetric]*
unfolding *sum-component[symmetric]*
unfolding *proj-onto-def proj-def[abs-def]*
unfolding *proj-onto-sum-rw*
by *vector*
qed
next
fix *i*
assume *col-not-eq: column i (Gram-Schmidt-column-k (Gram-Schmidt-upt-k A k) (Suc k)) ≠*
column (from-nat (Suc k)) (Gram-Schmidt-upt-k A k) -

$(\sum_{x \in \{\text{column } i \text{ (Gram-Schmidt-upt-}k \text{ } A \ k) \mid i. \text{ to-nat } i \leq k\}} (x \cdot \text{column (from-nat (Suc } k)) \text{ (Gram-Schmidt-upt-}k \text{ } A \ k) / (x \cdot x)) *_{\mathbb{R}} x)$
and $i: \text{to-nat } i \leq \text{Suc } k$
have $i\text{-not-suc-}k: i \neq \text{from-nat (Suc } k)$
proof (*rule ccontr, simp*)
assume $i2: i = \text{from-nat (Suc } k)$
have $\text{column } i \text{ (Gram-Schmidt-column-}k \text{ (Gram-Schmidt-upt-}k \text{ } A \ k) \text{ (Suc } k))$
 $=$
 $\text{column (from-nat (Suc } k)) \text{ (Gram-Schmidt-upt-}k \text{ } A \ k) -$
 $(\sum_{x \in \{\text{column } i \text{ (Gram-Schmidt-upt-}k \text{ } A \ k) \mid i. \text{ to-nat } i \leq k\}} (x \cdot \text{column (from-nat (Suc } k)) \text{ (Gram-Schmidt-upt-}k \text{ } A \ k) / (x \cdot x)) *_{\mathbb{R}} x)$
unfolding $i2$ *Gram-Schmidt-column-}k\text{-def column-def*
apply *auto*
unfolding *set-rw*
unfolding *vector-scale}R\text{-component[symmetric]*
unfolding *sum-component[symmetric]*
unfolding *proj-onto-def proj-def[abs-def]*
unfolding *proj-onto-sum-rw*
by *vector*
thus *False using col-not-eq by contradiction*
qed
show $\exists ia. \text{column } i \text{ (Gram-Schmidt-column-}k \text{ (Gram-Schmidt-upt-}k \text{ } A \ k) \text{ (Suc } k))$
 $= \text{column } ia \text{ (Gram-Schmidt-upt-}k \text{ } A \ k) \wedge \text{to-nat } ia \leq k$
proof (*rule exI[of - i], rule conjI, unfold Gram-Schmidt-upt-}k\text{-suc[symmetric], rule column-Gram-Schmidt-upt-}k\text{-preserves}*)
show $\text{to-nat } i < \text{Suc } k$ **using** i $i\text{-not-suc-}k$ **by** (*metis le-imp-less-or-eq from-nat-to-nat-id*)
thus $\text{to-nat } i \leq k$ **using** *less-Suc-eq-le* **by** *simp*
show $\text{Suc } k < \text{CARD('cols)}$ **using** k .
qed
qed
finally show *?thesis* .
qed

lemma *orthogonal-Gram-Schmidt-upt-}k:*
assumes $s: k < \text{ncols } A$
shows *pairwise orthogonal* ($\{\text{column } i \text{ (Gram-Schmidt-upt-}k \text{ } A \ k) \mid i. \text{to-nat } i \leq k\}$)
using s
proof (*induct k*)
case 0
have *set-rw:* $\{\text{column } i \text{ (Gram-Schmidt-upt-}k \text{ } A \ 0) \mid i. \text{to-nat } i \leq 0\} = \{\text{column } 0 \text{ (Gram-Schmidt-upt-}k \text{ } A \ 0)\}$
by (*simp add: to-nat-eq-0*)
show *?case* **unfolding** *set-rw* **unfolding** *pairwise-def* **by** *auto*
next

case (*Suc k*)
have *rw*: [$0..<Suc (Suc k)$] = [$0..<Suc k$] @ [(*Suc k*)] **by** *simp*
show ?*case*
unfolding *column-set-Gram-Schmidt-upt-k*[*OF Suc.prem*s[*unfolded ncols-def*],
of A]
unfolding *proj-onto-sum-rw*
by (*auto simp add: proj-def[symmetric] proj-onto-def[symmetric]*)
(*rule pairwise-orthogonal-proj-set, auto simp add: Suc.hyps Suc.prem*s *Suc-lessD*)
qed

lemma *columns-Gram-Schmidt-matrix-rw*:
{*column i (Gram-Schmidt-matrix A) | i. i ∈ UNIV*}
= {*column i (Gram-Schmidt-upt-k A (ncols A - 1)) | i. to-nat i ≤ (ncols A - 1)*}
proof (*auto*)
fix *i*
show $\exists ia. \text{column } i \text{ (Gram-Schmidt-matrix } A)$
= $\text{column } ia \text{ (Gram-Schmidt-upt-k } A \text{ (ncols } A - Suc \ 0)) \wedge \text{to-nat } ia \leq \text{ncols } A$
- $Suc \ 0$
apply (*rule exI[of - i]*) **unfolding** *Gram-Schmidt-matrix-def* **using** *to-nat-less-card*[*of*
i]
unfolding *ncols-def* **by** *fastforce*
show $\exists ia. \text{column } i \text{ (Gram-Schmidt-upt-k } A \text{ (ncols } A - Suc \ 0)) = \text{column } ia$
(*Gram-Schmidt-matrix A*)
unfolding *Gram-Schmidt-matrix-def* **by** *auto*
qed

corollary *orthogonal-Gram-Schmidt-matrix*:
shows *pairwise orthogonal* ({*column i (Gram-Schmidt-matrix A) | i. i ∈ UNIV*})
unfolding *columns-Gram-Schmidt-matrix-rw*
by (*rule orthogonal-Gram-Schmidt-upt-k, simp add: ncols-def*)

corollary *orthogonal-Gram-Schmidt-matrix2*:
shows *pairwise orthogonal* (*columns (Gram-Schmidt-matrix A)*)
unfolding *columns-def* **using** *orthogonal-Gram-Schmidt-matrix* .

lemma *column-Gram-Schmidt-column-k*:
fixes *A::'a::{real-inner} ^'n::{mod-type} ^'m::{mod-type}*
shows $\text{column } k \text{ (Gram-Schmidt-column-k } A \text{ (to-nat } k)) =$
($\text{column } k \text{ } A$) - ($\sum x \in \{\text{column } i \text{ } A \mid i. i < k\}. (x \cdot (\text{column } k \text{ } A) / (x \cdot x)) *_R x$)
unfolding *Gram-Schmidt-column-k-def* *column-def*
unfolding *from-nat-to-nat-id*
unfolding *proj-onto-def* *proj-def[abs-def]*
unfolding *proj-onto-sum-rw*
by *vector*

lemma *column-Gram-Schmidt-column-k'*:

fixes $A::'a::\{\text{real-inner}\}^{\wedge'n}::\{\text{mod-type}\}^{\wedge'm}::\{\text{mod-type}\}$
assumes $i\text{-not-}k: i \neq k$
shows $\text{column } i \text{ (Gram-Schmidt-column-}k \text{ } A \text{ (to-nat } k)) = (\text{column } i \text{ } A)$
using $i\text{-not-}k$
unfolding $\text{Gram-Schmidt-column-}k\text{-def column-def}$
unfolding $\text{from-nat-to-nat-id}$ **by** vector

definition $\text{cols-upt-}k \text{ } A \text{ } k = \{\text{column } i \text{ } A \mid i. i \leq \text{from-nat } k\}$

lemma $\text{cols-upt-}k\text{-insert}$:

fixes $A::'a^{\wedge'n}::\{\text{mod-type}\}^{\wedge'm}::\{\text{mod-type}\}$
assumes $k: (\text{Suc } k) < \text{ncols } A$
shows $\text{cols-upt-}k \text{ } A \text{ (Suc } k) = (\text{insert } (\text{column } (\text{from-nat } (\text{Suc } k)) \text{ } A) \text{ (cols-upt-}k \text{ } A \text{ } k))$
proof ($\text{unfold cols-upt-}k\text{-def, auto}$)
fix $i::'n$
assume $i: i \leq \text{from-nat } (\text{Suc } k)$ **and** $\text{column } i \text{ } A \neq \text{column } (\text{from-nat } (\text{Suc } k)) \text{ } A$
hence $i\text{-not-suc-}k: i \neq \text{from-nat } (\text{Suc } k)$ **by** auto
have $i\text{-le}: i \leq \text{from-nat } k$
proof –
have $i < \text{from-nat } (\text{Suc } k)$ **by** ($\text{metis le-imp-less-or-eq } i \text{ } i\text{-not-suc-}k$)
thus $?thesis$ **by** ($\text{metis Suc-eq-plus1 from-nat-suc le-Suc not-less}$)
qed
thus $\exists ia. \text{column } i \text{ } A = \text{column } ia \text{ } A \wedge ia \leq \text{from-nat } k$ **by** auto
next
fix $i::'n$
assume $i: i \leq \text{from-nat } k$
also have $\dots < \text{from-nat } (\text{Suc } k)$
by ($\text{rule from-nat-mono}[OF - k[\text{unfolded ncols-def}]]$, simp)
finally have $i \leq \text{from-nat } (\text{Suc } k)$ **by** simp
thus $\exists ia. \text{column } i \text{ } A = \text{column } ia \text{ } A \wedge ia \leq \text{from-nat } (\text{Suc } k)$ **by** auto
qed

lemma $\text{columns-eq-cols-upt-}k$:

fixes $A::'a^{\wedge'ncols}::\{\text{mod-type}\}^{\wedge'rows}::\{\text{mod-type}\}$
shows $\text{cols-upt-}k \text{ } A \text{ (ncols } A - 1) = \text{columns } A$
proof ($\text{unfold cols-upt-}k\text{-def columns-def, auto}$)
fix i
show $\exists ia. \text{column } i \text{ } A = \text{column } ia \text{ } A \wedge ia \leq \text{from-nat } (\text{ncols } A - \text{Suc } 0)$
proof ($\text{rule exI}[of - i]$, simp)
have $\text{to-nat } i < \text{ncols } A$ **using** $\text{to-nat-less-card}[of i]$ **unfolding** ncols-def **by** simp
hence $\text{to-nat } i \leq (\text{ncols } A - 1)$ **by** simp
hence $\text{to-nat } i \leq \text{to-nat } (\text{from-nat } (\text{ncols } A - 1)::'cols)$
using $\text{to-nat-from-nat-id}[of \text{ncols } A - 1]$, **where** $?'a='cols$ **unfolding** ncols-def
by simp

```

    thus  $i \leq \text{from-nat } (\text{ncols } A - \text{Suc } 0)$ 
      by (metis One-nat-def less-le-not-le linear to-nat-mono)
  qed
qed

lemma span-cols-upt-k-Gram-Schmidt-column-k:
  fixes  $A::'a::\{\text{real-inner}\}^{\wedge'n}::\{\text{mod-type}\}^{\wedge'm}::\{\text{mod-type}\}$ 
  assumes  $k < \text{ncols } A$ 
  and  $j < \text{ncols } A$ 
  shows  $\text{span } (\text{cols-upt-k } A \ k) = \text{span } (\text{cols-upt-k } (\text{Gram-Schmidt-column-k } A \ j) \ k)$ 
  using assms
proof (induct  $k$ )
  case 0
  have set-rw:  $\{\chi \text{ ia. } A \ \$ \ \text{ia} \ \$ \ i \mid i. \ i < 0\} = \{\}$  using least-mod-type not-less by
  auto
  have set-rw2:  $\{\text{column } i \ (\text{Gram-Schmidt-column-k } A \ j) \mid i. \ i \leq 0\} = \{\text{column } 0 \ (\text{Gram-Schmidt-column-k } A \ j)\}$ 
  by (auto, metis eq-iff least-mod-type)
  have set-rw3:  $\{\text{column } i \ A \mid i. \ i \leq 0\} = \{\text{column } 0 \ A\}$  by (auto, metis eq-iff
  least-mod-type)
  have col-0-eq:  $\text{column } 0 \ (\text{Gram-Schmidt-column-k } A \ j) = \text{column } 0 \ A$ 
  unfolding Gram-Schmidt-column-k-def column-def
  unfolding proj-onto-def proj-def[abs-def]
  by (simp add: set-rw)
  show ?case unfolding cols-upt-k-def from-nat-0 unfolding set-rw2 set-rw3 unfolding
  col-0-eq ..
next
  case (Suc  $k$ )
  have hyp:  $\text{span } (\text{cols-upt-k } A \ k) = \text{span } (\text{cols-upt-k } (\text{Gram-Schmidt-column-k } A \ j) \ k)$ 
  using Suc.prems Suc.hyps by auto
  have set-rw1:  $(\text{cols-upt-k } A \ (\text{Suc } k)) = \text{insert } (\text{column } (\text{from-nat } (\text{Suc } k)) \ A)$ 
  (cols-upt-k } A \ k)
  using cols-upt-k-insert
  by (auto intro!: cols-upt-k-insert[OF Suc.prems(1)])
  have set-rw2:  $(\text{cols-upt-k } (\text{Gram-Schmidt-column-k } A \ j) \ (\text{Suc } k)) =$ 
   $\text{insert } (\text{column } (\text{from-nat } (\text{Suc } k)) \ (\text{Gram-Schmidt-column-k } A \ j)) \ (\text{cols-upt-k}$ 
  (Gram-Schmidt-column-k } A \ j) \ k)
  using cols-upt-k-insert Suc.prems(1) unfolding ncols-def by auto
  show ?case
  proof (cases  $j = \text{Suc } k$ )
  case False
  have suc-not-k:  $\text{from-nat } (\text{Suc } k) \neq (\text{from-nat } j::'n)$ 
  proof (rule ccontr, simp)
  assume  $\text{from-nat } (\text{Suc } k) = (\text{from-nat } j::'n)$ 
  hence  $\text{Suc } k = j$  apply (rule from-nat-eq-imp-eq) using Suc.prems unfolding
  ncols-def .
  thus False using False by simp

```

```

qed
have tnfj: to-nat (from-nat j::'n) = j using to-nat-from-nat-id[OF Suc.prem(2)[unfolding
ncols-def]] .
let ?a-suc-k = column (from-nat (Suc k)) A
  have col-eq: column (from-nat (Suc k)) (Gram-Schmidt-column-k A j) =
?a-suc-k
  using column-Gram-Schmidt-column-k'[OF suc-not-k] unfolding tnfj .
  have k: k < CARD('n) using Suc.prem(1)[unfolding ncols-def] by simp
  show ?thesis unfolding set-rw1 set-rw2 col-eq unfolding span-insert unfolding
ing hyp ..
next
case True
define C where C = cols-upt-k A k
define B where B = cols-upt-k (Gram-Schmidt-column-k A j) k
define a where a = column (from-nat (Suc k)) A
let ?a=a - sum (λx. (x · a / (x · x)) *R x) C
let ?C=insert ?a C
  have col-rw: {column i A | i. i ≤ from-nat k} = {column i A | i. i < from-nat
(Suc k)}
  proof (auto)
    fix i::'n assume i: i ≤ from-nat k
    also have ... < from-nat (Suc k) by (rule from-nat-mono[OF - Suc.prem(1)[unfolding
ncols-def]], simp)
    finally show ∃ ia. column i A = column ia A ∧ ia < from-nat (Suc k) by
auto
  next
  fix i::'n
  assume i: i < from-nat (Suc k)
  hence i ≤ from-nat k unfolding Suc-eq-plus1 unfolding from-nat-suc by
(metis le-Suc not-less)
  thus ∃ ia. column i A = column ia A ∧ ia ≤ from-nat k by auto
qed
  have rw: column (from-nat (Suc k)) (Gram-Schmidt-column-k A j) = (a -
(∑x∈cols-upt-k A k. (x · a / (x · x)) *R x))
  unfolding Gram-Schmidt-column-k-def True unfolding cols-upt-k-def a-def
C-def
  unfolding column-def apply auto
  unfolding column-def[symmetric] col-rw minus-vec-def
  unfolding column-def vec-lambda-beta
  unfolding proj-onto-def proj-def[abs-def]
  unfolding proj-onto-sum-rw
  by auto
  have finite-C: finite C unfolding C-def cols-upt-k-def by auto
  {
    fix x b
    have th0: !!(a::'a~'m::{mod-type}) b c. a - (b - c) = c + (a - b)
      by (simp add: field-simps)
    have x - b *R (a - (∑x∈C. (x · a / (x · x)) *R x)) ∈ span C ↔ x -
b *R a ∈ span C

```

```

    apply (simp only: scaleR-right-diff-distrib th0)
    apply (rule span-add-eq)
    apply (rule span-mul)
    apply (rule span-sum)
    apply (rule span-mul)
    apply (rule span-base)
    apply assumption
  done
}
thus ?thesis unfolding set-eq-iff
unfolding C-def B-def unfolding set-rw1 unfolding set-rw2
unfolding span-breakdown-eq unfolding hyp
by (metis (mono-tags) B-def a-def rw)
qed
qed

```

```

corollary span-Gram-Schmidt-column-k:
fixes A::'a::{real-inner} ^'n::{mod-type} ^'m::{mod-type}
assumes k < ncols A
shows span (columns A) = span (columns (Gram-Schmidt-column-k A k))
unfolding columns-eq-cols-upt-k[symmetric]
using span-cols-upt-k-Gram-Schmidt-column-k[of ncols A - 1 A k]
using assms unfolding ncols-def by auto

```

```

corollary span-Gram-Schmidt-upt-k:
fixes A::'a::{real-inner} ^'n::{mod-type} ^'m::{mod-type}
assumes k < ncols A
shows span (columns A) = span (columns (Gram-Schmidt-upt-k A k))
using assms
proof (induct k)
case 0
have columns (Gram-Schmidt-column-k A 0) = columns A
proof (unfold columns-def, auto)
  fix i
  have set-rw: {χ ia. A $ ia $ i | i. i < from-nat 0} = {}
    by (auto, metis less-le-not-le least-mod-type from-nat-0)
  thus ∃ ia. column i (Gram-Schmidt-column-k A 0) = column ia A
    unfolding Gram-Schmidt-column-k-def column-def
    unfolding proj-onto-def proj-def[abs-def] by auto
  show ∃ ia. column i A = column ia (Gram-Schmidt-column-k A 0)
    using set-rw unfolding Gram-Schmidt-column-k-def column-def
    unfolding from-nat-0
    unfolding proj-onto-def proj-def[abs-def]
    by force
qed
thus ?case unfolding Gram-Schmidt-upt-k-def by auto
next

```

```

case (Suc k)
have hyp: span (columns A) = span (columns (Gram-Schmidt-upt-k A k))
  using Suc.premis Suc.hyps by auto
have span (columns (Gram-Schmidt-upt-k A (Suc k)))
  = span (columns (Gram-Schmidt-column-k (Gram-Schmidt-upt-k A k) (Suc k)))
  unfolding Gram-Schmidt-upt-k-suc ..
also have ... = span (columns (Gram-Schmidt-upt-k A k))
  using span-Gram-Schmidt-column-k[of Suc k (Gram-Schmidt-upt-k A k)]
  using Suc.premis unfolding ncols-def by auto
finally show ?case using hyp by auto
qed

corollary span-Gram-Schmidt-matrix:
  fixes A::'a::{real-inner} ^'n::{mod-type} ^'m::{mod-type}
  shows span (columns A) = span (columns (Gram-Schmidt-matrix A))
  unfolding Gram-Schmidt-matrix-def
  by (simp add: span-Gram-Schmidt-upt-k ncols-def)

lemma is-basis-columns-Gram-Schmidt-matrix:
  fixes A::real ^'n::{mod-type} ^'m::{mod-type}
  assumes b: is-basis (columns A)
  and c: card (columns A) = ncols A
  shows is-basis (columns (Gram-Schmidt-matrix A))
  ∧ card (columns (Gram-Schmidt-matrix A)) = ncols A
proof –
  have span-UNIV: vec.span (columns (Gram-Schmidt-matrix A)) = (UNIV::(real ^'m::{mod-type})
  set)
  using span-Gram-Schmidt-matrix b unfolding is-basis-def
  by (metis span-vec-eq)
  moreover have c-eq: card (columns (Gram-Schmidt-matrix A)) = ncols A
  proof –
  have card (columns A) ≤ card (columns (Gram-Schmidt-matrix A))
  by (metis b is-basis-def finite-columns vec.independent-span-bound span-UNIV
  top-greatest)
  thus ?thesis using c using card-columns-le-ncols by (metis le-antisym ncols-def)
  qed
  moreover have vec.independent (columns (Gram-Schmidt-matrix A))
  proof (rule vec.card-le-dim-spanning[of - UNIV::(real ^'m::{mod-type}) set])
  show columns (Gram-Schmidt-matrix A) ⊆ UNIV by simp
  show UNIV ⊆ vec.span (columns (Gram-Schmidt-matrix A)) using span-UNIV
  by auto
  show finite (columns (Gram-Schmidt-matrix A)) using finite-columns .
  show card (columns (Gram-Schmidt-matrix A)) ≤ vec.dim (UNIV::(real ^'m::{mod-type})
  set)
  by (metis b c c-eq eq-iff is-basis-def vec.dim-span-eq-card-independent)
  qed
  ultimately show ?thesis unfolding is-basis-def by simp
qed

```

From here on, we present some lemmas that will be useful for the formal-

sation of the QR decomposition.

lemma *column-gr-k-Gram-Schmidt-upt*:
fixes $A::\text{real}^{\wedge'n}::\{\text{mod-type}\}^{\wedge'm}::\{\text{mod-type}\}$
assumes $i > k$
and $i < \text{ncols } A$
shows $\text{column } (\text{from-nat } i) (\text{Gram-Schmidt-upt-k } A \ k) = \text{column } (\text{from-nat } i) \ A$
using *assms(1)*
proof (*induct k*)
assume $0 < i$
hence $(\text{from-nat } i::'n) \neq 0$
unfolding *from-nat-0[symmetric]* **using** *bij-from-nat[where ?'a='n]* **unfolding** *bij-betw-def*
by (*metis from-nat-eq-imp-eq gr-implies-not0 i ncols-def neq0-conv*)
thus $\text{column } (\text{from-nat } i) (\text{Gram-Schmidt-upt-k } A \ 0) = \text{column } (\text{from-nat } i) \ A$
unfolding *Gram-Schmidt-upt-k-def*
by (*simp add: Gram-Schmidt-column-k-def from-nat-0 column-def*)
next
case (*Suc k*)
have *hyp: column (from-nat i) (Gram-Schmidt-upt-k A k) = column (from-nat i) A*
using *Suc.hyps Suc.prem* **by** *auto*
have *to-nat-from-nat-suc-k: (to-nat (from-nat (Suc k)::'n)) = Suc k*
by (*metis Suc.prem dual-order.strict-trans from-nat-not-eq i ncols-def*)
have $\text{column } (\text{from-nat } i) (\text{Gram-Schmidt-upt-k } A \ (\text{Suc } k))$
 $= \text{column } (\text{from-nat } i) (\text{Gram-Schmidt-column-k } (\text{Gram-Schmidt-upt-k } A \ k)$
 $(\text{Suc } k))$
unfolding *Gram-Schmidt-upt-k-suc ..*
also have $\dots = \text{column } (\text{from-nat } i) (\text{Gram-Schmidt-upt-k } A \ k)$
proof (*rule column-Gram-Schmidt-column-k'*
 $[\text{of from-nat } i \text{ from-nat } (\text{Suc } k) (\text{Gram-Schmidt-upt-k } A \ k), \text{unfolding to-nat-from-nat-suc-k}]$)
show $\text{from-nat } i \neq (\text{from-nat } (\text{Suc } k)::'n)$
by (*metis Suc.prem not-less-iff-gr-or-eq from-nat-not-eq i ncols-def to-nat-from-nat-suc-k*)
qed
finally show *?case unfolding hyp .*
qed

lemma *columns-Gram-Schmidt-upt-k-rw*:
fixes $A::\text{real}^{\wedge'n}::\{\text{mod-type}\}^{\wedge'm}::\{\text{mod-type}\}$
assumes $k: \text{Suc } k < \text{ncols } A$
shows $\{\text{column } i (\text{Gram-Schmidt-upt-k } A \ (\text{Suc } k)) \mid i. i < \text{from-nat } (\text{Suc } k)\}$
 $= \{\text{column } i (\text{Gram-Schmidt-upt-k } A \ k) \mid i. i < \text{from-nat } (\text{Suc } k)\}$
proof (*auto*)
fix $i::'n$ **assume** $i < \text{from-nat } (\text{Suc } k)$
have *to-nat-from-nat-k: to-nat (from-nat (Suc k)::'n) = Suc k*
using *to-nat-from-nat-id k unfolding ncols-def* **by** *auto*
show $\exists ia. \text{column } i (\text{Gram-Schmidt-upt-k } A \ (\text{Suc } k)) = \text{column } ia (\text{Gram-Schmidt-upt-k } A \ k) \wedge ia < \text{from-nat } (\text{Suc } k)$
by (*metis column-Gram-Schmidt-upt-k-preserves i k ncols-def to-nat-le*)

show $\exists ia. \text{column } i \text{ (Gram-Schmidt-upt-} k \text{ } A \text{ } k) = \text{column } ia \text{ (Gram-Schmidt-upt-} k \text{ } A \text{ (Suc } k)) \wedge ia < \text{from-nat (Suc } k)$
by (*metis column-Gram-Schmidt-upt-k-preserves i k ncols-def to-nat-le*)
qed

lemma *column-Gram-Schmidt-upt-k:*

fixes $A::\text{real}^n::\{\text{mod-type}\}^m::\{\text{mod-type}\}$

assumes $k < \text{ncols } A$

shows $\text{column (from-nat } k) \text{ (Gram-Schmidt-upt-} k \text{ } A \text{ } k) =$

$(\text{column (from-nat } k) \text{ } A) - (\sum x \in \{\text{column } i \text{ (Gram-Schmidt-upt-} k \text{ } A \text{ } k) \mid i. i < \text{from-nat } k\}. (x \cdot (\text{column (from-nat } k) \text{ } A) / (x \cdot x)) *_R x)$

using *assms*

proof (*induct k, unfold from-nat-0*)

have $\text{set-rw: } \{\text{column } i \text{ (Gram-Schmidt-upt-} k \text{ } A \text{ } 0) \mid i. i < 0\} = \{\}$ **by** (*auto, metis least-mod-type not-le*)

have $\text{set-rw2: } \{\text{column } i \text{ } A \mid i. i < 0\} = \{\}$ **by** (*auto, metis least-mod-type not-le*)

have $\text{col-rw: column } 0 \text{ } A = \text{column } 0 \text{ (Gram-Schmidt-upt-} k \text{ } A \text{ } 0)$

unfolding *Gram-Schmidt-upt-k-def* **apply** *auto* **unfolding** *Gram-Schmidt-column-k-def from-nat-0*

unfolding *column-def*

using *set-rw2* **unfolding** *proj-onto-def proj-def[abs-def]*

by *vector*

show $\text{column } 0 \text{ (Gram-Schmidt-upt-} k \text{ } A \text{ } 0)$

$= \text{column } 0 \text{ } A - (\sum x \in \{\text{column } i \text{ (Gram-Schmidt-upt-} k \text{ } A \text{ } 0) \mid i. i < 0\}. (x \cdot \text{column } 0 \text{ } A / (x \cdot x)) *_R x)$

unfolding *set-rw col-rw* **by** *simp*

next

case (*Suc k*)

let $?ak = \text{column (from-nat } k) \text{ } A$

let $?uk = \text{column (from-nat } k) \text{ (Gram-Schmidt-upt-} k \text{ } A \text{ } k)$

let $?a-suck = \text{column (from-nat (Suc } k)) \text{ } A$

let $?u-suck = \text{column (from-nat (Suc } k)) \text{ (Gram-Schmidt-upt-} k \text{ } A \text{ (Suc } k))$

have $\text{to-nat-from-nat-k: to-nat (from-nat (Suc } k)::'n) = \text{(Suc } k)$

using *to-nat-from-nat-id Suc.prem* **unfolding** *ncols-def* **by** *auto*

have $\text{a-suc-rw: column (from-nat (Suc } k)) \text{ (Gram-Schmidt-upt-} k \text{ } A \text{ } k) = ?a-suck$

by (*rule column-gr-k-Gram-Schmidt-upt, auto simp add: Suc.prem*)

have $\text{set-rw: } \{\text{column } i \text{ (Gram-Schmidt-upt-} k \text{ } A \text{ (Suc } k)) \mid i. i < \text{from-nat (Suc } k)\}$

$= \{\text{column } i \text{ (Gram-Schmidt-upt-} k \text{ } A \text{ } k) \mid i. i < \text{from-nat (Suc } k)\}$

by (*rule columns-Gram-Schmidt-upt-k-rw[OF Suc.prem]*)

have $?u-suck = \text{column (from-nat (Suc } k)) \text{ (Gram-Schmidt-upt-} k \text{ } A \text{ } k) -$

$(\sum x \in \{\text{column } i \text{ (Gram-Schmidt-upt-} k \text{ } A \text{ } k) \mid i.$

$i < \text{from-nat (Suc } k)\}. (x \cdot \text{column (from-nat (Suc } k)) \text{ (Gram-Schmidt-upt-} k \text{ } A \text{ } k) / (x \cdot x)) *_R x)$

unfolding *Gram-Schmidt-upt-k-suc*

using *column-Gram-Schmidt-column-k[of from-nat (Suc k) (Gram-Schmidt-upt-k A k)]*

unfolding *to-nat-from-nat-k* **by** *auto*

also have ... = ?a-suck -
 $(\sum x \in \{\text{column } i \text{ (Gram-Schmidt-upt-} k \text{ } A \text{)} \mid i.$
 $i < \text{from-nat (Suc } k)\}. (x \cdot ?a\text{-suck} / (x \cdot x)) *_R x)$ **unfolding** a-suc-rw ..
finally show ?u-suck = ?a-suck - $(\sum x \in \{\text{column } i \text{ (Gram-Schmidt-upt-} k \text{ } A \text{ (Suc$
 $k)\} \mid i.$
 $i < \text{from-nat (Suc } k)\}. (x \cdot ?a\text{-suck} / (x \cdot x)) *_R x)$
unfolding set-rw .
qed

lemma column-Gram-Schmidt-upt-k-preserves2:

fixes A::real[^]'n::{mod-type}[^]'m::{mod-type}
assumes a≤(from-nat i)
and i ≤ j
and j < ncols A
shows column a (Gram-Schmidt-upt-k A i) = column a (Gram-Schmidt-upt-k A
j)
using assms
proof (induct j)
case 0
show column a (Gram-Schmidt-upt-k A i) = column a (Gram-Schmidt-upt-k A
0) **by** (metis 0.prem(2) le-0-eq)
next
case (Suc j)
show ?case
proof (cases a=from-nat (Suc j))
case False **note** a-not-suc-j=False
have rw1: (to-nat (from-nat (Suc j)::'n)) = Suc j
using to-nat-from-nat-id Suc.prem(2) **unfolding** ncols-def **by** auto
show ?thesis **unfolding** Gram-Schmidt-upt-k-suc **using** column-Gram-Schmidt-column-k'[OF
a-not-suc-j] **unfolding** rw1
by (metis Gram-Schmidt-upt-k-suc Suc.hyps Suc.prem(2) Suc.prem(3)
Suc-le-lessD assms(1) le-Suc-eq nat-less-le)
next
case True
have (from-nat i::'n) ≤ from-nat (Suc j) **by** (rule from-nat-mono'[OF Suc.prem(2)
Suc.prem(3)[unfolded ncols-def]])
hence from-nat i = (from-nat (Suc j)::'n) **using** Suc.prem(1) **unfolding** True
by simp
hence i-eq-suc: i=Suc j **apply** (rule from-nat-eq-imp-eq) **using** Suc.prem
unfolding ncols-def **by** auto
show ?thesis **unfolding** True i-eq-suc ..
qed
qed

lemma set-columns-Gram-Schmidt-matrix:

```

fixes  $A::\text{real}^n::\{\text{mod-type}\}^m::\{\text{mod-type}\}$ 
shows  $\{\text{column } i \text{ (Gram-Schmidt-matrix } A)|i. i < k\} = \{\text{column } i \text{ (Gram-Schmidt-upt-} k$ 
 $A \text{ (to-nat } k))|i. i < k\}$ 
proof (auto)
  fix  $i$  assume  $i < k$ 
  show  $\exists ia. \text{column } i \text{ (Gram-Schmidt-matrix } A) = \text{column } ia \text{ (Gram-Schmidt-upt-} k$ 
 $A \text{ (to-nat } k)) \wedge ia < k$ 
  proof (rule exI[of - i], rule conjI)
    show  $\text{column } i \text{ (Gram-Schmidt-matrix } A) = \text{column } i \text{ (Gram-Schmidt-upt-} k$ 
 $A \text{ (to-nat } k))$ 
    unfolding Gram-Schmidt-matrix-def
    proof (rule column-Gram-Schmidt-upt-k-preserves2[symmetric])
      show  $i \leq \text{from-nat (to-nat } k)$  using  $i$  unfolding from-nat-to-nat-id by auto
      show  $\text{to-nat } k \leq \text{ncols } A - 1$  unfolding ncols-def using to-nat-less-card[of
 $k$ ] by auto
      show  $\text{ncols } A - 1 < \text{ncols } A$  unfolding ncols-def by simp
    qed
    show  $i < k$  using  $i$  .
  qed
show  $\exists ia. \text{column } i \text{ (Gram-Schmidt-upt-} k A \text{ (to-nat } k)) = \text{column } ia \text{ (Gram-Schmidt-matrix}$ 
 $A) \wedge ia < k$ 
proof (rule exI[of - i], rule conjI)
  show  $\text{column } i \text{ (Gram-Schmidt-upt-} k A \text{ (to-nat } k)) = \text{column } i \text{ (Gram-Schmidt-matrix}$ 
 $A)$ 
  unfolding Gram-Schmidt-matrix-def
  proof (rule column-Gram-Schmidt-upt-k-preserves2)
    show  $i \leq \text{from-nat (to-nat } k)$  using  $i$  unfolding from-nat-to-nat-id by auto
    show  $\text{to-nat } k \leq \text{ncols } A - 1$  unfolding ncols-def using to-nat-less-card[of
 $k$ ] by auto
    show  $\text{ncols } A - 1 < \text{ncols } A$  unfolding ncols-def by simp
  qed
  show  $i < k$  using  $i$  .
qed
qed

```

lemma column-Gram-Schmidt-matrix:

```

fixes  $A::\text{real}^n::\{\text{mod-type}\}^m::\{\text{mod-type}\}$ 
shows  $\text{column } k \text{ (Gram-Schmidt-matrix } A)$ 
 $= (\text{column } k A) - (\sum x \in \{\text{column } i \text{ (Gram-Schmidt-matrix } A)|i. i < k\}. (x \cdot$ 
 $(\text{column } k A) / (x \cdot x)) *_R x)$ 
proof -
  have  $k: \text{to-nat } k < \text{ncols } A$  using to-nat-less-card[of  $k$ ] unfolding ncols-def by
simp
  have  $\text{column } k \text{ (Gram-Schmidt-matrix } A) = \text{column } k \text{ (Gram-Schmidt-upt-} k$ 
 $A \text{ (ncols } A - 1))$ 
  unfolding Gram-Schmidt-matrix-def ..
  also have  $\dots = \text{column } k \text{ (Gram-Schmidt-upt-} k A \text{ (to-nat } k))$ 

```

proof (rule column-Gram-Schmidt-upt-k-preserves2[symmetric])
show $k \leq \text{from-nat } (to\text{-nat } k)$ **unfolding** from-nat-to-nat-id ..
show $to\text{-nat } k \leq \text{ncols } A - 1$ **unfolding** ncols-def **using** to-nat-less-card[where
 $?'a='n]$
by (metis le-diff-conv2 add-leE less-diff-conv less-imp-le-nat less-le-not-le
nat-le-linear suc-not-zero to-nat-plus-one-less-card')
show $\text{ncols } A - 1 < \text{ncols } A$ **unfolding** ncols-def **by** auto
qed
also have $\dots = \text{column } k \ A - (\sum_{x \in \{\text{column } i \ (Gram\text{-Schmidt-upt-}k \ A \ (to\text{-nat } k)) \mid i. i < k\}} (x \cdot \text{column } k \ A / (x \cdot x)) *_R x)$
using column-Gram-Schmidt-upt-k[OF k] **unfolding** from-nat-to-nat-id **by** auto
also have $\dots = \text{column } k \ A - (\sum_{x \in \{\text{column } i \ (Gram\text{-Schmidt-matrix } A) \mid i. i < k\}} (x \cdot \text{column } k \ A / (x \cdot x)) *_R x)$
unfolding set-columns-Gram-Schmidt-matrix[symmetric] ..
finally show ?thesis .
qed

corollary column-Gram-Schmidt-matrix2:
fixes $A::\text{real}^n::\{\text{mod-type}\}^m::\{\text{mod-type}\}$
shows $(\text{column } k \ A) = \text{column } k \ (Gram\text{-Schmidt-matrix } A)$
 $+ (\sum_{x \in \{\text{column } i \ (Gram\text{-Schmidt-matrix } A) \mid i. i < k\}} (x \cdot (\text{column } k \ A) / (x \cdot x)) *_R x)$
using column-Gram-Schmidt-matrix[of k A] **by** simp

lemma independent-columns-Gram-Schmidt-matrix:
fixes $A::\text{real}^n::\{\text{mod-type}\}^m::\{\text{mod-type}\}$
assumes $b: \text{vec.independent } (\text{columns } A)$
and $c: \text{card } (\text{columns } A) = \text{ncols } A$
shows $\text{vec.independent } (\text{columns } (Gram\text{-Schmidt-matrix } A)) \wedge \text{card } (\text{columns } (Gram\text{-Schmidt-matrix } A)) = \text{ncols } A$
using $b \ c \ \text{card-columns-le-ncols } \text{vec.card-eq-dim-span-indep } \text{vec.dim-span-eq-iff } \text{finite-columns}$
 $\text{vec.independent-span-bound } \text{ncols-def } \text{span-Gram-Schmidt-matrix}$
by (metis (no-types, lifting) vec.card-ge-dim-independent vec.dim-span-eq-card-independent span-vec-eq)

lemma column-eq-Gram-Schmidt-matrix:
fixes $A::\text{real}^n::\{\text{mod-type}\}^m::\{\text{mod-type}\}$
assumes $r: \text{rank } A = \text{ncols } A$
and $c: \text{column } i \ (Gram\text{-Schmidt-matrix } A) = \text{column } ia \ (Gram\text{-Schmidt-matrix } A)$
shows $i = ia$
proof (rule ccontr)
assume $i\text{-not-}ia: i \neq ia$
have $\text{columns } (Gram\text{-Schmidt-matrix } A) = (\lambda x. \text{column } x \ (Gram\text{-Schmidt-matrix } A))$

```

A)) ‘ (UNIV::('n::{mod-type}) set)
  unfolding columns-def by auto
  also have ... = (λx. column x (Gram-Schmidt-matrix A)) ‘ ((UNIV::('n::{mod-type})
set) - {ia})
  proof (unfold image-def, auto)
    fix xa
    show ∃ x ∈ UNIV - {ia}. column xa (Gram-Schmidt-matrix A) = column x
(Gram-Schmidt-matrix A)
    proof (cases xa = ia)
      case True thus ?thesis using c i-not-ia by (metis DiffI UNIV-I empty-iff
insert-iff)
    next
      case False thus ?thesis by auto
    qed
  qed
  finally have columns-rw: columns (Gram-Schmidt-matrix A) = (λx. column x
(Gram-Schmidt-matrix A)) ‘ (UNIV - {ia}) .
  have ncols A = card (columns (Gram-Schmidt-matrix A))
  by (metis full-rank-imp-is-basis2 independent-columns-Gram-Schmidt-matrix r)
  also have ... ≤ card (UNIV - {ia}) unfolding columns-rw by (rule card-image-le,
simp)
  also have ... = card (UNIV::'n set) - 1 by (simp add: card-Diff-singleton)
  finally show False unfolding ncols-def
  by (metis Nat.add-0-right le-diff-conv2 One-nat-def Suc-n-not-le-n add-Suc-right
one-le-card-finite)
qed

```

```

lemma scaleR-columns-Gram-Schmidt-matrix:
  fixes A::real^'n::{mod-type} ^'m::{mod-type}
  assumes i ≠ j
  and rank A = ncols A
  shows column j (Gram-Schmidt-matrix A) · column i (Gram-Schmidt-matrix A)
= 0
proof -
  have column j (Gram-Schmidt-matrix A) ≠ column i (Gram-Schmidt-matrix A)
  using column-eq-Gram-Schmidt-matrix assms by auto
  thus ?thesis using orthogonal-Gram-Schmidt-matrix2 unfolding pairwise-def
orthogonal-def columns-def
  by blast
qed

```

3.1.4 Examples of execution

Code lemma

```
lemmas Gram-Schmidt-step-def[unfolded proj-onto-def proj-def[abs-def],code]
```

```

value let a = map (list-to-vec::real list=> real^4) [[4,-2,-1,2],
[-6,3,4,-8], [5,-5,-3,-4]] in
map vec-to-list (Gram-Schmidt a)

```

```

value let a = map (list-to-vec::real list=> real^4) [[4,-2,-1,2],
  [-6,3,4,-8], [5,-5,-3,-4]] in
  map vec-to-list (Gram-Schmidt2 a)

```

```

value let A = list-of-list-to-matrix [[4,-2,-1,2],
  [-6,3,4,-8], [5,-5,-3,-4]]::real^4^3 in
  matrix-to-list-of-list (Gram-Schmidt-matrix A)

```

```

end

```

4 QR Decomposition

```

theory QR-Decomposition
imports Gram-Schmidt
begin

```

4.1 The QR Decomposition of a matrix

First of all, it's worth noting what an orthogonal matrix is. In linear algebra, an orthogonal matrix is a square matrix with real entries whose columns and rows are orthogonal unit vectors.

Although in some texts the QR decomposition is presented over square matrices, it can be applied to any matrix. There are some variants of the algorithm, depending on the properties that the output matrices satisfy (see for instance, http://inst.eecs.berkeley.edu/~ee127a/book/login/1_mats_qr.html). We present two of them below.

Let A be a matrix with m rows and n columns (A is $m \times n$).

Case 1: Starting with a matrix whose column rank is maximum. We can define the QR decomposition to obtain:

- $A = Q ** R$.
- Q has m rows and n columns. Its columns are orthogonal unit vectors and *Finite-Cartesian-Product.transpose* $Q * Q = \text{mat } (1::'a)$. In addition, if A is a square matrix, then Q will be an orthonormal matrix.
- R is $n \times n$, invertible and upper triangular.

Case 2: The called full QR decomposition. We can obtain:

- $A = Q ** R$
- Q is an orthogonal matrix (Q is $m \times m$).
- R is $m \times n$ and upper triangular, but it isn't invertible.

We have decided to formalise the first one, because it's the only useful for solving the linear least squares problem (<http://math.mit.edu/linearalgebra/ila0403.pdf>).

If we have an unsolvable system $A *v x = b$, we can try to find an approximate solution. A plausible choice (not the only one) is to seek an x with the property that $\|A ** x - y\|$ (the magnitude of the error) is as small as possible. That x is the least squares approximation.

We will demonstrate that the best approximation (the solution for the linear least squares problem) is the x that satisfies:

$$(\text{transpose } A) ** A *v x = (\text{transpose } A) *v b$$

Now we want to compute that x .

If we are working with the first case, A can be substituted by $Q**R$ and then obtain the solution of the least squares approximation by means of the QR decomposition:

$$x = (\text{inverse } R)**(\text{transpose } Q) *v b$$

On the contrary, if we are working with the second case after substituting A by $Q**R$ we obtain:

$$(\text{transpose } R) ** R *v x = (\text{transpose } R) ** (\text{transpose } Q) *v b$$

But the R matrix is not invertible (so neither is $\text{transpose } R$). The left part of the equation $(\text{transpose } R) ** R$ is not going to be an upper triangular matrix, so it can't either be solved using backward-substitution.

4.1.1 Divide a vector by its norm

An orthogonal matrix is a matrix whose rows (and columns) are orthonormal vectors. So, in order to obtain the QR decomposition, we have to normalise (divide by the norm) the vectors obtained with the Gram-Schmidt algorithm.

definition *divide-by-norm* $A = (\chi \text{ a b. normalize (column b } A) \$ a)$

Properties

lemma *norm-column-divide-by-norm*:

fixes $A::'a::\{\text{real-inner}\}^{\wedge} \text{cols}^{\wedge} \text{rows}$

assumes a : column a $A \neq 0$

shows $\text{norm (column } a \text{ (divide-by-norm } A)) = 1$

proof –

have *not-0*: $\text{norm } (\chi \text{ i. } A \$ i \$ a) \neq 0$ **by** (*metis a column-def norm-eq-zero*)

have $\text{column } a \text{ (divide-by-norm } A) = (\chi \text{ i. } (1 / \text{norm } (\chi \text{ i. } A \$ i \$ a)) *_R A \$ i \$ a)$

unfolding *divide-by-norm-def column-def normalize-def* **by** *auto*

also have $\dots = (1 / \text{norm } (\chi \text{ i. } A \$ i \$ a)) *_R (\chi \text{ i. } A \$ i \$ a)$

unfolding *vec-eq-iff* **by** *auto*

finally have $\text{norm (column } a \text{ (divide-by-norm } A)) = \text{norm } ((1 / \text{norm } (\chi \text{ i. } A \$ i \$ a)) *_R (\chi \text{ i. } A \$ i \$ a))$

```

    by simp
  also have ... = |1 / norm (χ i. A $ i $ a)| * norm (χ i. A $ i $ a)
    unfolding norm-scaleR ..
  also have ... = (1 / norm (χ i. A $ i $ a)) * norm (χ i. A $ i $ a)
    by auto
  also have ... = 1 using not-0 by auto
  finally show ?thesis .
qed

lemma span-columns-divide-by-norm:
  shows span (columns A) = span (columns (divide-by-norm A))
  unfolding real-vector.span-eq
proof (auto)
  fix x assume x: x ∈ columns (divide-by-norm A)
  from this obtain i where x-col-i: x=column i (divide-by-norm A) unfolding
  columns-def by blast
  also have ... = (1/norm (column i A)) *R (column i A)
    unfolding divide-by-norm-def column-def normalize-def by vector
  finally have x-eq: x=(1/norm (column i A)) *R (column i A) .
  show x ∈ span (columns A)
    by (unfold x-eq, rule span-mul, rule span-base, auto simp add: columns-def)
next
  fix x
  assume x: x ∈ columns A
  show x ∈ span (columns (divide-by-norm A))
  proof (cases x=0)
    case True show ?thesis by (metis True span-0)
  next
    case False
    from x obtain i where x-col-i: x=column i A unfolding columns-def by blast
    have x=column i A using x-col-i .
    also have ... = norm (column i A) *R column i (divide-by-norm A)
      using False unfolding x-col-i columns-def divide-by-norm-def column-def
  normalize-def by vector
    finally have x-eq: x = norm (column i A) *R column i (divide-by-norm A) .
    show x ∈ span (columns (divide-by-norm A))
      by (unfold x-eq, rule span-mul, rule span-base,
        auto simp add: columns-def Let-def)
  qed
qed

```

Code lemmas

```

definition divide-by-norm-row A a = vec-lambda(% b. ((1 / norm (column b A))
*_R column b A) $ a)

```

```

lemma divide-by-norm-row-code[code abstract]:
  vec-nth (divide-by-norm-row A a) = (% b. ((1 / norm (column b A)) *_R column
b A) $ a)
  unfolding divide-by-norm-row-def by (metis (lifting) vec-lambda-beta)

```

```

lemma divide-by-norm-code [code abstract]:
  vec-nth (divide-by-norm A) = divide-by-norm-row A
  unfolding divide-by-norm-def unfolding divide-by-norm-row-def[abs-def]
  unfolding normalize-def
  by fastforce

```

4.1.2 The QR Decomposition

The QR decomposition. Given a real matrix A , the algorithm will return a pair (Q, R) where Q is a matrix whose columns are orthogonal unit vectors, R is upper triangular and $A = Q ** R$.

definition *QR-decomposition* $A = (\text{let } Q = \text{divide-by-norm } (\text{Gram-Schmidt-matrix } A) \text{ in } (Q, (\text{transpose } Q) ** A))$

lemma *is-basis-columns-fst-QR-decomposition*:

```

fixes  $A::\text{real}^n::\{\text{mod-type}\}^m::\{\text{mod-type}\}$ 
assumes  $b$ : is-basis (columns A)
and  $c$ : card (columns A) = ncols A
shows is-basis (columns (fst (QR-decomposition A)))
   $\wedge$  card (columns (fst (QR-decomposition A))) = ncols A
proof (rule conjI, unfold is-basis-def, rule conjI)
  have vec.span (columns (fst (QR-decomposition A))) = vec.span (columns (Gram-Schmidt-matrix
A))
    unfolding vec.span-eq
  proof (auto)
    fix  $x$  show  $x \in \text{vec.span } (\text{columns } (\text{Gram-Schmidt-matrix } A))$ 
      using assms(1) assms(2) is-basis-columns-Gram-Schmidt-matrix is-basis-def
by auto
    next
      fix  $x$ 
      assume  $x$ :  $x \in \text{columns } (\text{Gram-Schmidt-matrix } A)$ 
      from this obtain  $i$  where  $x\text{-col-}i$ :  $x = \text{column } i \text{ } (\text{Gram-Schmidt-matrix } A)$ 
unfolding columns-def by blast
      have zero-not-in:  $x \neq 0$  using is-basis-columns-Gram-Schmidt-matrix[OF b c]
unfolding is-basis-def
      using vec.dependent-zero[of (columns (Gram-Schmidt-matrix A))]  $x$  by auto
      have  $x = \text{column } i \text{ } (\text{Gram-Schmidt-matrix } A)$  using  $x\text{-col-}i$  .
      also have  $\dots = \text{norm } (\text{column } i \text{ } (\text{Gram-Schmidt-matrix } A)) *_R \text{column } i$ 
(divide-by-norm (Gram-Schmidt-matrix A))
      using zero-not-in unfolding  $x\text{-col-}i$  columns-def divide-by-norm-def column-def
normalize-def by vector
      finally have  $x\text{-eq}$ :  $x = \text{norm } (\text{column } i \text{ } (\text{Gram-Schmidt-matrix } A)) *_R \text{column } i$ 
(divide-by-norm (Gram-Schmidt-matrix A)) .
      show  $x \in \text{vec.span } (\text{columns } (\text{fst } (\text{QR-decomposition } A)))$ 
        unfolding  $x\text{-eq}$  span-vec-eq
        apply (rule subspace-mul)
        apply (auto simp add: columns-def QR-decomposition-def Let-def subspace-span
intro: span-superset)

```



```

    using span-superset by force
  qed
  thus s: vec.span (columns (fst (QR-decomposition A))) = (UNIV::(realm::{mod-type})
  set)
    using is-basis-columns-Gram-Schmidt-matrix[OF b c] unfolding is-basis-def
  by simp
  thus card (columns (fst (QR-decomposition A))) = ncols A
  by (metis (opaque-lifting, mono-tags) b c card-columns-le-ncols vec.card-le-dim-spanning

    finite-columns vec.indep-card-eq-dim-span is-basis-def ncols-def top-greatest)
  thus vec.independent (columns (fst (QR-decomposition A)))
  by (metis s b c vec.card-eq-dim-span-indep finite-columns vec.indep-card-eq-dim-span
  is-basis-def)
  qed

```

lemma *orthogonal-fst-QR-decomposition:*

```

  shows pairwise orthogonal (columns (fst (QR-decomposition A)))
  unfolding pairwise-def columns-def
  proof (auto)
    fix i ia
    assume col-not-eq: column i (fst (QR-decomposition A)) ≠ column ia (fst (QR-decomposition
  A))
    hence i-not-ia: i ≠ ia by auto
    from col-not-eq obtain a
    where (fst (QR-decomposition A)) $ a $ i ≠ (fst (QR-decomposition A)) $ a $
  ia
    unfolding column-def by force
    hence col-not-eq2: (column i (Gram-Schmidt-matrix A)) ≠ (column ia (Gram-Schmidt-matrix
  A))
    using col-not-eq unfolding QR-decomposition-def Let-def fst-conv
    by (metis (lifting) divide-by-norm-def vec-lambda-beta)
    have d1: column i (fst (QR-decomposition A))
    = (1 / norm (χ ia. Gram-Schmidt-matrix A $ ia $ i)) *R (column i (Gram-Schmidt-matrix
  A))
    unfolding QR-decomposition-def Let-def fst-conv
    unfolding divide-by-norm-def column-def normalize-def unfolding vec-eq-iff
  by auto
    have d2: column ia (fst (QR-decomposition A))
    = (1 / norm (χ i. Gram-Schmidt-matrix A $ i $ ia)) *R (column ia (Gram-Schmidt-matrix
  A))
    unfolding QR-decomposition-def Let-def fst-conv
    unfolding divide-by-norm-def column-def normalize-def unfolding vec-eq-iff
  by auto
    show orthogonal (column i (fst (QR-decomposition A))) (column ia (fst (QR-decomposition
  A)))
    unfolding d1 d2 apply (rule orthogonal-mult) using orthogonal-Gram-Schmidt-matrix[of
  A]
    unfolding pairwise-def using col-not-eq2 by auto

```

qed

lemma *qk-uk-norm*:

($1 / (\text{norm } (\text{column } k \text{ } ((\text{Gram-Schmidt-matrix } A))))$) *_R ($\text{column } k \text{ } ((\text{Gram-Schmidt-matrix } A))$)
= $\text{column } k \text{ } (\text{fst } (\text{QR-decomposition } A))$
unfolding *QR-decomposition-def Let-def fst-conv divide-by-norm-def*
unfolding *column-def normalize-def by vector*

lemma *norm-columns-fst-QR-decomposition*:

fixes $A :: \text{real}^n :: \{\text{mod-type}\}^m :: \{\text{mod-type}\}$
assumes $\text{rank } A = \text{ncols } A$
shows $\text{norm } (\text{column } i \text{ } (\text{fst } (\text{QR-decomposition } A))) = 1$

proof –

have $\text{vec.independent } (\text{columns } (\text{Gram-Schmidt-matrix } A))$
by (*metis assms full-rank-imp-is-basis2 independent-columns-Gram-Schmidt-matrix*)
hence $\text{column } i \text{ } (\text{Gram-Schmidt-matrix } A) \neq 0$
using $\text{vec.dependent-zero[of columns } (\text{Gram-Schmidt-matrix } A)]$
unfolding *columns-def by auto*
thus $\text{norm } (\text{column } i \text{ } (\text{fst } (\text{QR-decomposition } A))) = 1$
unfolding *QR-decomposition-def Let-def fst-conv*
by (*rule norm-column-divide-by-norm*)

qed

corollary *span-fst-QR-decomposition*:

fixes $A :: \text{real}^n :: \{\text{mod-type}\}^m :: \{\text{mod-type}\}$
shows $\text{vec.span } (\text{columns } A) = \text{vec.span } (\text{columns } (\text{fst } (\text{QR-decomposition } A)))$
unfolding *span-Gram-Schmidt-matrix[of A]*
unfolding *QR-decomposition-def Let-def fst-conv*
by (*metis span-columns-divide-by-norm span-vec-eq*)

corollary *col-space-QR-decomposition*:

fixes $A :: \text{real}^n :: \{\text{mod-type}\}^m :: \{\text{mod-type}\}$
shows $\text{col-space } A = \text{col-space } (\text{fst } (\text{QR-decomposition } A))$
unfolding *col-space-def using span-fst-QR-decomposition*
by *auto*

lemma *independent-columns-fst-QR-decomposition*:

fixes $A :: \text{real}^n :: \{\text{mod-type}\}^m :: \{\text{mod-type}\}$
assumes $b : \text{vec.independent } (\text{columns } A)$
and $c : \text{card } (\text{columns } A) = \text{ncols } A$
shows $\text{vec.independent } (\text{columns } (\text{fst } (\text{QR-decomposition } A)))$
 $\wedge \text{card } (\text{columns } (\text{fst } (\text{QR-decomposition } A))) = \text{ncols } A$

```

proof –
  have  $r$ :  $\text{rank } A = \text{ncols } A$  thm is-basis-imp-full-rank
  proof –
    have  $\text{rank } A = \text{col-rank } A$  unfolding rank-col-rank ..
    also have  $\dots = \text{vec.dim } (\text{col-space } A)$  unfolding col-rank-def ..
    also have  $\dots = \text{card } (\text{columns } A)$ 
      unfolding col-space-def using b
      by (rule vec.dim-span-eq-card-independent)
    also have  $\dots = \text{ncols } A$  using  $c$  .
    finally show ?thesis .
  qed
  have  $\text{vec.independent } (\text{columns } (\text{fst } (\text{QR-decomposition } A)))$ 
    by (metis b c col-rank-def col-space-QR-decomposition col-space-def
      full-rank-imp-is-basis2 vec.indep-card-eq-dim-span ncols-def rank-col-rank)
  moreover have  $\text{card } (\text{columns } (\text{fst } (\text{QR-decomposition } A))) = \text{ncols } A$ 
    by (metis col-space-QR-decomposition full-rank-imp-is-basis2 ncols-def r rank-eq-dim-col-space')
  ultimately show ?thesis by simp
qed

```

```

lemma orthogonal-matrix-fst-QR-decomposition:
  fixes  $A::\text{real}^n::\{\text{mod-type}\}^m::\{\text{mod-type}\}$ 
  assumes  $r$ :  $\text{rank } A = \text{ncols } A$ 
  shows  $\text{transpose } (\text{fst } (\text{QR-decomposition } A)) ** (\text{fst } (\text{QR-decomposition } A)) =$ 
 $\text{mat } 1$ 
proof (unfold vec-eq-iff, clarify, unfold mat-1-fun, auto)
  define  $Q$  where  $Q = \text{fst } (\text{QR-decomposition } A)$ 
  have  $n$ :  $\forall i. \text{norm } (\text{column } i \ Q) = 1$  unfolding  $Q$ -def using norm-columns-fst-QR-decomposition[OF r] by auto
  have  $c$ :  $\text{card } (\text{columns } Q) = \text{ncols } A$  unfolding  $Q$ -def
    by (metis full-rank-imp-is-basis2 independent-columns-fst-QR-decomposition r)
  have  $p$ : pairwise orthogonal ( $\text{columns } Q$ ) by (metis Q-def orthogonal-fst-QR-decomposition)
  fix  $ia$ 
  have  $(\text{transpose } Q ** Q) \$ ia \$ ia = \text{column } ia \ Q \cdot \text{column } ia \ Q$ 
    unfolding matrix-matrix-mult-inner-mult unfolding row-transpose ..
  also have  $\dots = 1$  using  $n$  norm-eq-1 by blast
  finally show  $(\text{transpose } Q ** Q) \$ ia \$ ia = 1$  .
  fix  $i$ 
  assume  $i$ -not- $ia$ :  $i \neq ia$ 
  have  $\text{column-}i$ -not- $ia$ :  $\text{column } i \ Q \neq \text{column } ia \ Q$ 
  proof (rule ccontr, simp)
    assume  $\text{col-}i$ - $ia$ :  $\text{column } i \ Q = \text{column } ia \ Q$ 
    have  $rw$ :  $(\lambda i. \text{column } i \ Q)' (\text{UNIV} - \{ia\}) = \{\text{column } i \ Q \mid i. i \neq ia\}$  unfolding
 $\text{columns-def}$  by auto
    have  $\text{card } (\text{columns } Q) = \text{card } (\{\text{column } i \ Q \mid i. i \neq ia\})$ 
      by (rule bij-betw-same-card[of id], unfold bij-betw-def columns-def, auto, metis
 $\text{col-}i$ - $ia$   $i$ -not- $ia$ )
    also have  $\dots = \text{card } ((\lambda i. \text{column } i \ Q)' (\text{UNIV} - \{ia\}))$  unfolding  $rw$  ..
    also have  $\dots \leq \text{card } (\text{UNIV} - \{ia\})$  by (metis card-image-le finite-code)

```

```

    also have ... < CARD ('n) by simp
    finally show False using c unfolding ncols-def by simp
qed
hence oia: orthogonal (column i Q) (column ia Q)
  using p unfolding pairwise-def unfolding columns-def by auto
have (transpose Q ** Q) $ i $ ia = column i Q · column ia Q
  unfolding matrix-matrix-mult-inner-mult unfolding row-transpose ..
also have ... = 0 using oia unfolding orthogonal-def .
finally show (transpose Q ** Q) $ i $ ia = 0 .
qed

corollary orthogonal-matrix-fst-QR-decomposition':
  fixes A::real^'n::{mod-type} ^'n::{mod-type}
  assumes rank A = ncols A
  shows orthogonal-matrix (fst (QR-decomposition A))
  by (metis assms orthogonal-matrix orthogonal-matrix-fst-QR-decomposition)

lemma column-eq-fst-QR-decomposition:
  fixes A::real^'n::{mod-type} ^'m::{mod-type}
  assumes r: rank A = ncols A
  and c: column i (fst (QR-decomposition A)) = column ia (fst (QR-decomposition
A))
  shows i = ia
proof (rule ccontr)
  assume i-not-ia: i ≠ ia
  have columns (fst (QR-decomposition A)) = (λx. column x (fst (QR-decomposition
A))) ' (UNIV::('n::{mod-type}) set)
    unfolding columns-def by auto
  also have ... = (λx. column x (fst (QR-decomposition A))) ' ((UNIV::('n::{mod-type})
set) - {ia})
  proof (unfold image-def, auto)
    fix xa
    show ∃ x∈UNIV - {ia}. column xa (fst (QR-decomposition A)) = column x
(fst (QR-decomposition A))
    proof (cases xa = ia)
      case True thus ?thesis using c i-not-ia by (metis DiffI UNIV-I empty-iff
insert-iff)
    next
      case False thus ?thesis by auto
    qed
  qed
  finally have columns-rw: columns (fst (QR-decomposition A))
    = (λx. column x (fst (QR-decomposition A))) ' (UNIV - {ia}) .
  have ncols A = card (columns (fst (QR-decomposition A)))
  by (metis full-rank-imp-is-basis2 independent-columns-fst-QR-decomposition r)
also have ... ≤ card (UNIV - {ia}) unfolding columns-rw by (rule card-image-le,
simp)
also have ... = card (UNIV::'n set) - 1 by (simp add: card-Diff-singleton)

```

finally show *False unfolding ncols-def*
by (*metis Nat.add-0-right le-diff-conv2 One-nat-def Suc-n-not-le-n add-Suc-right one-le-card-finite*)
qed

corollary *column-QR-decomposition:*

fixes $A::\text{real}^n::\{\text{mod-type}\}^m::\{\text{mod-type}\}$
assumes $r:\text{rank } A = \text{ncols } A$
shows $\text{column } k ((\text{Gram-Schmidt-matrix } A))$
 $= (\text{column } k A) - (\sum_{x \in \{\text{column } i (\text{fst } (QR\text{-decomposition } A)) \mid i < k\}} (x \cdot (\text{column } k A) / (x \cdot x)) *_R x)$
proof –
let $?uk = \text{column } k ((\text{Gram-Schmidt-matrix } A))$
let $?qk = \text{column } k (\text{fst } (QR\text{-decomposition } A))$
let $?ak = (\text{column } k A)$
define f **where** $f x = (1/\text{norm } x) *_R x$ **for** $x :: \text{real}^m::\{\text{mod-type}\}$
let $?g = \lambda x::\text{real}^m::\{\text{mod-type}\}. (x \cdot (\text{column } k A) / (x \cdot x)) *_R x$
have $\text{set-rw}: \{\text{column } i (\text{fst } (QR\text{-decomposition } A)) \mid i < k\} = f \{ \text{column } i ((\text{Gram-Schmidt-matrix } A)) \mid i < k \}$
proof (*auto*)
fix i
assume $i: i < k$
have $\text{col-rw}: \text{column } i (\text{fst } (QR\text{-decomposition } A)) = (1/\text{norm } (\text{column } i ((\text{Gram-Schmidt-matrix } A)))) *_R (\text{column } i ((\text{Gram-Schmidt-matrix } A)))$
unfolding *QR-decomposition-def Let-def fst-conv divide-by-norm-def column-def normalize-def* **by** *vector*
thus $\text{column } i (\text{fst } (QR\text{-decomposition } A)) \in f \{ \text{column } i ((\text{Gram-Schmidt-matrix } A)) \mid i < k \}$
unfolding *f-def* **using** i
by *auto*
show $\exists ia. f (\text{column } i ((\text{Gram-Schmidt-matrix } A))) = \text{column } ia (\text{fst } (QR\text{-decomposition } A)) \wedge ia < k$
by (*rule exI[of - i], simp add: f-def col-rw i*)
qed
have $(\sum_{x \in \{\text{column } i (\text{fst } (QR\text{-decomposition } A)) \mid i < k\}} (x \cdot ?ak / (x \cdot x)) *_R x)$
 $= (\sum_{x \in (f \{ \text{column } i ((\text{Gram-Schmidt-matrix } A)) \mid i < k \})} (x \cdot ?ak / (x \cdot x)) *_R x)$
unfolding *set-rw ..*
also have $\dots = \text{sum } (?g \circ f) \{ \text{column } i ((\text{Gram-Schmidt-matrix } A)) \mid i < k \}$
proof (*rule sum.reindex, unfold inj-on-def, auto*)
fix i **assume** $i: i < k$ **and** $ia: ia < k$
and $f\text{-eq}: f (\text{column } i ((\text{Gram-Schmidt-matrix } A))) = f (\text{column } ia ((\text{Gram-Schmidt-matrix } A)))$
have $f_i: f (\text{column } i ((\text{Gram-Schmidt-matrix } A))) = \text{column } i (\text{fst } (QR\text{-decomposition } A))$
unfolding *f-def QR-decomposition-def Let-def fst-conv divide-by-norm-def column-def normalize-def*

by *vector*
have $fia: f(\text{column } ia \text{ (Gram-Schmidt-matrix } A)) = \text{column } ia \text{ (fst (QR-decomposition } A))$
unfolding *f-def QR-decomposition-def Let-def fst-conv divide-by-norm-def column-def normalize-def*
by *vector*
have $i = ia$ **using** *column-eq-fst-QR-decomposition[OF r]* **f-eq** **unfolding** *fi fia*
by *simp*
thus $\text{column } i \text{ (Gram-Schmidt-matrix } A) = \text{column } ia \text{ (Gram-Schmidt-matrix } A)$ **by** *simp*
qed
also **have** $\dots = (\sum_{x \in \{\text{column } i \text{ (Gram-Schmidt-matrix } A) \mid i. i < k\}} (1 / \text{norm } x) *_{\mathbb{R}} x \cdot ?ak / ((1 / \text{norm } x) *_{\mathbb{R}} x \cdot (1 / \text{norm } x) *_{\mathbb{R}} x)) *_{\mathbb{R}} (1 / \text{norm } x) *_{\mathbb{R}} x$ **unfolding** *o-def f-def ..*
also **have** $\dots = (\sum_{x \in \{\text{column } i \text{ (Gram-Schmidt-matrix } A) \mid i. i < k\}} (1 / \text{norm } x) *_{\mathbb{R}} x \cdot ?ak) *_{\mathbb{R}} (1 / \text{norm } x) *_{\mathbb{R}} x$
proof (*rule sum.cong, simp*)
fix x **assume** $x: x \in \{\text{column } i \text{ (Gram-Schmidt-matrix } A) \mid i. i < k\}$
have *vec.independent* $\{\text{column } i \text{ (Gram-Schmidt-matrix } A) \mid i. i < k\}$
proof (*rule vec.independent-mono[of columns (Gram-Schmidt-matrix A)]*)
show *vec.independent* $(\text{columns (Gram-Schmidt-matrix } A))$
using *full-rank-imp-is-basis2[of (Gram-Schmidt-matrix A)]*
by (*metis full-rank-imp-is-basis2 independent-columns-Gram-Schmidt-matrix*
r)
show $\{\text{column } i \text{ (Gram-Schmidt-matrix } A) \mid i. i < k\} \subseteq \text{columns (Gram-Schmidt-matrix } A)$
unfolding *columns-def* **by** *auto*
qed
hence $x \neq 0$ **using** *vec.dependent-zero[of \{\text{column } i \text{ (Gram-Schmidt-matrix } A) \mid i. i < k\} x]*
by *blast*
hence $((1 / \text{norm } x) *_{\mathbb{R}} x \cdot (1 / \text{norm } x) *_{\mathbb{R}} x) = 1$ **by** (*metis inverse-eq-divide norm-eq-1 norm-sgn sgn-div-norm*)
thus $((1 / \text{norm } x) *_{\mathbb{R}} x \cdot ?ak / ((1 / \text{norm } x) *_{\mathbb{R}} x \cdot (1 / \text{norm } x) *_{\mathbb{R}} x)) *_{\mathbb{R}} (1 / \text{norm } x) *_{\mathbb{R}} x = ((1 / \text{norm } x) *_{\mathbb{R}} x \cdot \text{column } k \text{ } A) *_{\mathbb{R}} (1 / \text{norm } x) *_{\mathbb{R}} x$ **by** *auto*
qed
also **have** $\dots = (\sum_{x \in \{\text{column } i \text{ (Gram-Schmidt-matrix } A) \mid i. i < k\}} ((x \cdot ?ak) / (x \cdot x)) *_{\mathbb{R}} x)$
proof (*rule sum.cong, simp*)
fix x
assume $x: x \in \{\text{column } i \text{ (Gram-Schmidt-matrix } A) \mid i. i < k\}$
show $((1 / \text{norm } x) *_{\mathbb{R}} x \cdot \text{column } k \text{ } A) *_{\mathbb{R}} (1 / \text{norm } x) *_{\mathbb{R}} x = (x \cdot \text{column } k \text{ } A / (x \cdot x)) *_{\mathbb{R}} x$
by (*metis (opaque-lifting, no-types) mult.right-neutral inner-commute inner-scaleR-right norm-cauchy-schwarz-eq scaleR-one scaleR-scaleR times-divide-eq-right times-divide-times-eq*)
qed

finally have $?ak - (\sum x \in \{\text{column } i \text{ (fst (QR-decomposition } A)) \mid i. i < k\}. (x \cdot ?ak / (x \cdot x)) *_R x)$
 $= ?ak - (\sum x \in \{\text{column } i \text{ (Gram-Schmidt-matrix } A) \mid i. i < k\}. ((x \cdot ?ak) / (x \cdot x)) *_R x)$ **by auto**
also have ... $= ?uk$ **using** *column-Gram-Schmidt-matrix[of k A]* **by auto**
finally show *?thesis ..*
qed

lemma *column-QR-decomposition'*:
fixes $A::\text{real}^n::\{\text{mod-type}\}^m::\{\text{mod-type}\}$
assumes $r: \text{rank } A = \text{ncols } A$
shows $(\text{column } k \text{ } A) = \text{column } k \text{ ((Gram-Schmidt-matrix } A))$
 $+ (\sum x \in \{\text{column } i \text{ (fst (QR-decomposition } A)) \mid i. i < k\}. (x \cdot (\text{column } k \text{ } A) / (x \cdot x)) *_R x)$
using *column-QR-decomposition[OF r]* **by simp**

lemma *norm-uk-eq*:
fixes $A::\text{real}^n::\{\text{mod-type}\}^m::\{\text{mod-type}\}$
assumes $r: \text{rank } A = \text{ncols } A$
shows $\text{norm} (\text{column } k \text{ ((Gram-Schmidt-matrix } A))) = ((\text{column } k \text{ (fst(QR-decomposition } A))) \cdot (\text{column } k \text{ } A))$
proof –
let $?uk = \text{column } k \text{ ((Gram-Schmidt-matrix } A))$
let $?qk = \text{column } k \text{ (fst(QR-decomposition } A))$
let $?ak = (\text{column } k \text{ } A)$
have $\text{sum-rw}: (?uk \cdot (\sum x \in \{\text{column } i \text{ (Gram-Schmidt-matrix } A) \mid i. i < k\}. (x \cdot ?ak / (x \cdot x)) *_R x)) = 0$
proof –
have $(?uk \cdot (\sum x \in \{\text{column } i \text{ (Gram-Schmidt-matrix } A) \mid i. i < k\}. (x \cdot ?ak / (x \cdot x)) *_R x))$
 $= ((\sum x \in \{\text{column } i \text{ (Gram-Schmidt-matrix } A) \mid i. i < k\}. ?uk \cdot ((x \cdot ?ak / (x \cdot x)) *_R x)))$
unfolding *inner-sum-right ..*
also have ... $= (\sum x \in \{\text{column } i \text{ (Gram-Schmidt-matrix } A) \mid i. i < k\}. ((x \cdot ?ak / (x \cdot x)) * (?uk \cdot x)))$
unfolding *inner-scaleR-right ..*
also have ... $= 0$
proof (*rule sum.neutral, clarify*)
fix $x \ i$ **assume** $i < k$
hence $?uk \cdot \text{column } i \text{ (Gram-Schmidt-matrix } A) = 0$
by (*metis less-irrefl r scaleR-columns-Gram-Schmidt-matrix*)
thus $\text{column } i \text{ (Gram-Schmidt-matrix } A) \cdot ?ak / (\text{column } i \text{ (Gram-Schmidt-matrix } A) \cdot \text{column } i \text{ (Gram-Schmidt-matrix } A)) *$
 $(?uk \cdot \text{column } i \text{ (Gram-Schmidt-matrix } A)) = 0$ **by auto**
qed
finally show *?thesis .*
qed
have $?qk \cdot ?ak = ((1/(\text{norm } ?uk)) *_R ?uk) \cdot ?ak$ **unfolding** *qk-uk-norm ..*

also have ... = $(1/(\text{norm } ?uk)) * (?uk \cdot ?ak)$ **unfolding inner-scaleR-left ..**
also have ... =
 $(1/(\text{norm } ?uk)) * (?uk \cdot (?uk + (\sum x \in \{\text{column } i \text{ (Gram-Schmidt-matrix } A) \mid i. i < k\}. (x \cdot ?ak / (x \cdot x)) *_R x)))$
using column-Gram-Schmidt-matrix2[of k A] by auto
also have ... = $(1/(\text{norm } ?uk)) * ((?uk \cdot ?uk) + (?uk \cdot (\sum x \in \{\text{column } i \text{ (Gram-Schmidt-matrix } A) \mid i. i < k\}. (x \cdot ?ak / (x \cdot x)) *_R x)))$
unfolding inner-add-right ..
also have ... = $(1/(\text{norm } ?uk)) * (?uk \cdot ?uk)$ **unfolding sum-rw by auto**
also have ... = $\text{norm } ?uk$
by (*metis abs-of-nonneg divide-eq-imp div-by-0 inner-commute inner-ge-zero inner-real-def*
norm-mult-vec real-inner-1-right real-norm-def times-divide-eq-right)
finally show *?thesis ..*
qed

corollary column-QR-decomposition2:
fixes $A::\text{real}^n::\{\text{mod-type}\}^m::\{\text{mod-type}\}$
assumes $r: \text{rank } A = \text{ncols } A$
shows (*column k A*)
 $= (\sum x \in \{\text{column } i \text{ (fst (QR-decomposition } A)) \mid i. i \leq k\}. (x \cdot (\text{column } k \text{ } A)) *_R x)$
proof –
let $?uk = \text{column } k \text{ (Gram-Schmidt-matrix } A)$
let $?qk = \text{column } k \text{ (fst (QR-decomposition } A))$
let $?ak = (\text{column } k \text{ } A)$
have $\text{set-rw}: \{\text{column } i \text{ (fst (QR-decomposition } A)) \mid i. i \leq k\}$
 $= \text{insert } (\text{column } k \text{ (fst (QR-decomposition } A))) \{\text{column } i \text{ (fst (QR-decomposition } A)) \mid i. i < k\}$
by (*auto, metis less-linear not-less*)
have $uk\text{-norm-uk-qk}: ?uk = \text{norm } ?uk *_R ?qk$
proof –
have $\text{vec.independent } (\text{columns } (\text{Gram-Schmidt-matrix } A))$
by (*metis full-rank-imp-is-basis2 independent-columns-Gram-Schmidt-matrix*
r)
moreover have $?uk \in \text{columns } (\text{Gram-Schmidt-matrix } A)$ **unfolding columns-def**
by auto
ultimately have $?uk \neq 0$
using $\text{vec.dependent-zero[of columns } (\text{Gram-Schmidt-matrix } A)]$ **unfolding**
columns-def by auto
hence $\text{norm-not-0}: \text{norm } ?uk \neq 0$ **unfolding norm-eq-zero .**
have $\text{norm } (?uk) *_R ?qk = (\text{norm } ?uk) *_R ((1 / \text{norm } ?uk) *_R ?uk)$ **using**
qk-uk-norm[of k A] by simp
also have ... = $((\text{norm } ?uk) * (1 / \text{norm } ?uk)) *_R ?uk$ **unfolding scaleR-scaleR**
..
also have ... = $?uk$ **using norm-not-0 by auto**
finally show *?thesis ..*
qed
have $\text{norm-qk-1}: ?qk \cdot ?qk = 1$

using *norm-eq-1 norm-columns-fst-QR-decomposition*[*OF r*]
by *auto*
have $?ak = ?uk + (\sum x \in \{\text{column } i \text{ (fst (QR-decomposition } A)) \mid i. i < k\}. (x \cdot ?ak / (x \cdot x)) *_R x)$
using *column-QR-decomposition'*[*OF r*] **by** *auto*
also have $\dots = (\text{norm } ?uk *_R ?qk) + (\sum x \in \{\text{column } i \text{ (fst (QR-decomposition } A)) \mid i. i < k\}. (x \cdot ?ak / (x \cdot x)) *_R x)$
using *uk-norm-uk-qk* **by** *simp*
also have $\dots = ((?qk \cdot ?ak) *_R ?qk) + (\sum x \in \{\text{column } i \text{ (fst (QR-decomposition } A)) \mid i. i < k\}. (x \cdot ?ak / (x \cdot x)) *_R x)$
unfolding *norm-uk-eq*[*OF r*] ..
also have $\dots = ((?qk \cdot ?ak) / (?qk \cdot ?qk)) *_R ?qk + (\sum x \in \{\text{column } i \text{ (fst (QR-decomposition } A)) \mid i. i < k\}. (x \cdot ?ak / (x \cdot x)) *_R x)$
using *norm-qk-1* **by** *fastforce*
also have $\dots = (\sum x \in \text{insert } ?qk \{\text{column } i \text{ (fst (QR-decomposition } A)) \mid i. i < k\}. (x \cdot ?ak / (x \cdot x)) *_R x)$
proof (*rule sum.insert[symmetric]*)
show *finite* $\{\text{column } i \text{ (fst (QR-decomposition } A)) \mid i. i < k\}$ **by** *simp*
show $\text{column } k \text{ (fst (QR-decomposition } A)) \notin \{\text{column } i \text{ (fst (QR-decomposition } A)) \mid i. i < k\}$
proof (*rule ccontr, simp*)
assume $\exists i. \text{column } k \text{ (fst (QR-decomposition } A)) = \text{column } i \text{ (fst (QR-decomposition } A)) \wedge i < k$
from *this* **obtain** *i* **where** *col-eq*: $\text{column } k \text{ (fst (QR-decomposition } A)) = \text{column } i \text{ (fst (QR-decomposition } A))$
and *i-less-k*: $i < k$ **by** *blast*
show *False* **using** *column-eq-fst-QR-decomposition*[*OF r col-eq*] *i-less-k* **by** *simp*
qed
qed
also have $\dots = (\sum x \in \{\text{column } i \text{ (fst (QR-decomposition } A)) \mid i. i \leq k\}. (x \cdot (\text{column } k \text{ } A)) *_R x)$
proof (*rule sum.cong, simp add: set-rw*)
fix *x* **assume** $x \in \{\text{column } i \text{ (fst (QR-decomposition } A)) \mid i. i \leq k\}$
from *this* **obtain** *i* **where** *i*: $x = \text{column } i \text{ (fst (QR-decomposition } A))$ **by** *blast*
hence $(x \cdot x) = 1$ **using** *norm-eq-1 norm-columns-fst-QR-decomposition*[*OF r*]
by *auto*
thus $(x \cdot \text{column } k \text{ } A) / (x \cdot x) *_R x = (x \cdot \text{column } k \text{ } A) *_R x$ **by** *simp*
qed
finally show *?thesis* .
qed

lemma *orthogonal-columns-fst-QR-decomposition*:

assumes *i-not-ia*: $(\text{column } i \text{ (fst (QR-decomposition } A))) \neq (\text{column } ia \text{ (fst (QR-decomposition } A)))$
shows $(\text{column } i \text{ (fst (QR-decomposition } A)) \cdot \text{column } ia \text{ (fst (QR-decomposition } A))) = 0$

proof –
have i : $\text{column } i \text{ (fst (QR-decomposition } A)) \in \text{columns (fst (QR-decomposition } A))$ **unfolding** *columns-def* **by** *auto*
have ia : $\text{column } ia \text{ (fst (QR-decomposition } A)) \in \text{columns (fst (QR-decomposition } A))$ **unfolding** *columns-def* **by** *auto*
show *?thesis*
using *orthogonal-fst-QR-decomposition[of A] i ia i-not-ia* **unfolding** *pair-wise-def orthogonal-def*
by *auto*
qed

lemma *scaler-column-fst-QR-decomposition*:

fixes $A::\text{real}^n::\{\text{mod-type}\}^m::\{\text{mod-type}\}$
assumes $i: i > j$
and r : $\text{rank } A = \text{ncols } A$
shows $\text{column } i \text{ (fst (QR-decomposition } A)) \cdot \text{column } j \text{ } A = 0$

proof –
have $\text{column } i \text{ (fst (QR-decomposition } A)) \cdot \text{column } j \text{ } A$
 $= \text{column } i \text{ (fst (QR-decomposition } A)) \cdot (\sum_{x \in \{\text{column } i \text{ (fst (QR-decomposition } A)) | i. i \leq j\}} (x \cdot (\text{column } j \text{ } A)) *_R x)$
using *column-QR-decomposition2[OF r]* **by** *presburger*
also $\text{have } \dots = (\sum_{x \in \{\text{column } i \text{ (fst (QR-decomposition } A)) | i. i \leq j\}} (x \cdot (\text{column } j \text{ } A)) *_R x)$ **unfolding** *real-inner-class.inner-sum-right ..*
also $\text{have } \dots = (\sum_{x \in \{\text{column } i \text{ (fst (QR-decomposition } A)) | i. i \leq j\}} (x \cdot (\text{column } j \text{ } A)) * (\text{column } i \text{ (fst (QR-decomposition } A)) \cdot x))$ **unfolding** *real-inner-class.inner-scaleR-right ..*
also $\text{have } \dots = 0$
proof (*rule sum.neutral, clarify*)
fix ia **assume** $ia: ia \leq j$
have *i-not-ia*: $i \neq ia$ **using** i **by** *simp*
hence $(\text{column } i \text{ (fst (QR-decomposition } A)) \neq \text{column } ia \text{ (fst (QR-decomposition } A)))$
by (*metis column-eq-fst-QR-decomposition r*)
hence $(\text{column } i \text{ (fst (QR-decomposition } A)) \cdot \text{column } ia \text{ (fst (QR-decomposition } A))) = 0$
by (*rule orthogonal-columns-fst-QR-decomposition*)
thus $\text{column } ia \text{ (fst (QR-decomposition } A)) \cdot \text{column } j \text{ } A * (\text{column } i \text{ (fst (QR-decomposition } A)) \cdot \text{column } ia \text{ (fst (QR-decomposition } A))) = 0$
by *auto*
qed
finally **show** *?thesis* .
qed

lemma *R-Qi-Aj*:

fixes $A::\text{real}^n::\{\text{mod-type}\}^m::\{\text{mod-type}\}$
shows $(\text{snd (QR-decomposition } A)) \$ i \$ j = \text{column } i \text{ (fst (QR-decomposition } A)) \cdot \text{column } j \text{ } A$
unfolding *QR-decomposition-def Let-def snd-conv matrix-matrix-mult-inner-mult*

unfolding *row-transpose by auto*

lemma *sums-columns-Q-0:*

fixes $A::\text{real}^n::\{\text{mod-type}\}^m::\{\text{mod-type}\}$

assumes $r: \text{rank } A = \text{ncols } A$

shows $(\sum x \in \{\text{column } i \text{ (fst (QR-decomposition } A)) \mid i. i > b\}. x \cdot \text{column } b \text{ } A * x$
 $\$ a) = 0$

proof (*rule sum.neutral, auto*)

fix i **assume** $b < i$

thus $\text{column } i \text{ (fst (QR-decomposition } A)) \cdot \text{column } b \text{ } A = 0$

by (*rule scalar-column-fst-QR-decomposition, simp add: r*)

qed

lemma *QR-decomposition-mult:*

fixes $A::\text{real}^n::\{\text{mod-type}\}^m::\{\text{mod-type}\}$

assumes $r: \text{rank } A = \text{ncols } A$

shows $A = (\text{fst (QR-decomposition } A)) ** (\text{snd (QR-decomposition } A))$

proof –

have $\forall b. \text{column } b \text{ } A = \text{column } b \text{ ((fst (QR-decomposition } A)) ** (\text{snd (QR-decomposition } A)))$

proof (*clarify*)

fix b

have $(\text{fst (QR-decomposition } A)) ** (\text{snd (QR-decomposition } A))$

$= (\chi \ i \ j. \sum k \in \text{UNIV}. \text{fst (QR-decomposition } A) \ \$ \ i \ \$ \ k * (\text{column } k \text{ (fst (QR-decomposition } A)) \cdot \text{column } j \text{ } A))$

unfolding *matrix-matrix-mult-def R-Qi-Aj by auto*

hence $\text{column } b \text{ ((fst (QR-decomposition } A)) ** (\text{snd (QR-decomposition } A))) =$

$\text{column } b \text{ ((}\chi \ i \ j. \sum k \in \text{UNIV}. \text{fst (QR-decomposition } A) \ \$ \ i \ \$ \ k * (\text{column } k \text{ (fst (QR-decomposition } A)) \cdot \text{column } j \text{ } A))$

by *auto*

also have $\dots = (\sum x \in \{\text{column } i \text{ (fst (QR-decomposition } A)) \mid i. i \leq b\}. (x \cdot \text{column } b \text{ } A) *_R x)$

proof (*subst column-def, subst vec-eq-iff, auto*)

fix a

define f **where** $f \ i = \text{column } i \text{ (fst (QR-decomposition } A))$ **for** i

define g **where** $g \ x = (\text{THE } i. x = \text{column } i \text{ (fst (QR-decomposition } A)))$

for x

have $f\text{-eq: } f' \text{UNIV} = \{\text{column } i \text{ (fst (QR-decomposition } A)) \mid i. i \in \text{UNIV}\}$

unfolding *f-def by auto*

have $\text{inj-f: inj } f$

by (*metis inj-on-def f-def column-eq-fst-QR-decomposition r*)

have $(\sum x \in \{\text{column } i \text{ (fst (QR-decomposition } A)) \mid i. i \leq b\}. x \cdot \text{column } b \text{ } A$
 $* x \ \$ \ a)$

$= (\sum x \in \{\text{column } i \text{ (fst (QR-decomposition } A)) \mid i. i \in \text{UNIV}\}. x \cdot \text{column } b$
 $A * x \ \$ \ a)$

proof –

let $?c = \{\text{column } i \text{ (fst (QR-decomposition } A)) \mid i. i \in \text{UNIV}\}$

```

let ?d = {column i (fst (QR-decomposition A)) | i. i ≤ b}
let ?f = {column i (fst (QR-decomposition A)) | i. i > b}
have set-rw: ?c = ?d ∪ ?f by force
have (∑ x ∈ ?c. x · column b A * x $ a)
  = (∑ x ∈ (?d ∪ ?f). x · column b A * x $ a) using set-rw by simp
also have ... = (∑ x ∈ ?d. x · column b A * x $ a) + (∑ x ∈ ?f. x · column
b A * x $ a)
  by (rule sum.union-disjoint, auto, metis f-def inj-eq inj-f not-le)
also have ... = (∑ x ∈ ?d. x · column b A * x $ a) using sums-columns-Q-0[OF
r] by auto
  finally show ?thesis ..
qed
also have ... = (∑ x ∈ f'UNIV. x · column b A * x $ a) using f-eq by auto
also have ... = (∑ k ∈ UNIV. fst (QR-decomposition A) $ a $ k * (column k
(fst (QR-decomposition A)) · column b A))
  unfolding sum.reindex[OF inj-f] unfolding f-def column-def by (rule
sum.cong, simp-all)
  finally show (∑ k ∈ UNIV. fst (QR-decomposition A) $ a $ k * (column k
(fst (QR-decomposition A)) · column b A)) =
  (∑ x ∈ {column i (fst (QR-decomposition A)) | i. i ≤ b}. x · column b A * x
$ a) ..
qed
also have ... = column b A
  using column-QR-decomposition2[OF r] by simp
  finally show column b A = column b (fst (QR-decomposition A)) ** snd
(QR-decomposition A) ..
qed
thus ?thesis unfolding column-def vec-eq-iff by auto
qed

```

lemma upper-triangular-snd-QR-decomposition:

```

fixes A :: real ^ 'n :: {mod-type} ^ 'm :: {mod-type}
assumes r: rank A = ncols A
shows upper-triangular (snd (QR-decomposition A))
proof (unfold upper-triangular-def, auto)
  fix i j :: 'n
  assume j-less-i: j < i
  have snd (QR-decomposition A) $ i $ j = column i (fst (QR-decomposition A))
· column j A
  unfolding QR-decomposition-def Let-def fst-conv snd-conv
  unfolding matrix-matrix-mult-inner-mult row-transpose ..
  also have ... = 0 using scaler-column-fst-QR-decomposition[OF j-less-i r] .
  finally show snd (QR-decomposition A) $ i $ j = 0 by auto
qed

```

lemma upper-triangular-invertible:

```

fixes A :: real ^ 'n :: {finite, wellorder} ^ 'n :: {finite, wellorder}

```

assumes u : upper-triangular A
and d : $\forall i. A \ \$ \ i \ \$ \ i \neq 0$
shows invertible A
proof –
have $det\text{-}R$: $det \ A = (prod \ (\lambda i. \ A \ \$ \ i \ \$ \ i) \ (UNIV::'n \ set))$
using $det\text{-}upperdiagonal \ u$ **unfolding** $upper\text{-}triangular\text{-}def$ **by** $blast$
also **have** $\dots \neq 0$ **using** d **by** $auto$
finally **show** $?thesis$ **by** $(metis \ invertible\text{-}det\text{-}nz)$
qed

lemma $invertible\text{-}snd\text{-}QR\text{-}decomposition$:
fixes $A::real^{n::\{mod\text{-}type\}}^{m::\{mod\text{-}type\}}$
assumes r : $rank \ A = ncols \ A$
shows invertible $(snd \ (QR\text{-}decomposition \ A))$
proof $(rule \ upper\text{-}triangular\text{-}invertible)$
show upper-triangular $(snd \ (QR\text{-}decomposition \ A))$
using $upper\text{-}triangular\text{-}snd\text{-}QR\text{-}decomposition[OF \ r]$.
show $\forall i. \ snd \ (QR\text{-}decomposition \ A) \ \$ \ i \ \$ \ i \neq 0$
proof $(rule \ allI)$
fix i
have ind : $vec.independent \ (columns \ (Gram\text{-}Schmidt\text{-}matrix \ A))$
by $(metis \ full\text{-}rank\text{-}imp\text{-}is\text{-}basis2$
 $independent\text{-}columns\text{-}Gram\text{-}Schmidt\text{-}matrix \ r)$
hence $zero\text{-}not\text{-}in$: $0 \notin \ (columns \ (Gram\text{-}Schmidt\text{-}matrix \ A))$ **by** $(metis \ vec.independent\text{-}zero)$
hence c : $column \ i \ (Gram\text{-}Schmidt\text{-}matrix \ A) \neq 0$ **unfolding** $columns\text{-}def$ **by**
 $simp$
have $snd \ (QR\text{-}decomposition \ A) \ \$ \ i \ \$ \ i = column \ i \ (fst \ (QR\text{-}decomposition \ A))$
 $\cdot \ column \ i \ A$
unfolding $QR\text{-}decomposition\text{-}def \ Let\text{-}def \ snd\text{-}conv \ fst\text{-}conv$
unfolding $matrix\text{-}matrix\text{-}mult\text{-}inner\text{-}mult$
unfolding $row\text{-}transpose \ ..$
also **have** $\dots = norm \ (column \ i \ (Gram\text{-}Schmidt\text{-}matrix \ A))$
unfolding $norm\text{-}uk\text{-}eq[OF \ r, \ symmetric] \ ..$
also **have** $\dots \neq 0$ **by** $(rule \ ccontr, \ simp \ add: \ c)$
finally **show** $snd \ (QR\text{-}decomposition \ A) \ \$ \ i \ \$ \ i \neq 0$.
qed
qed

lemma $QR\text{-}decomposition$:
fixes $A::real^{n::\{mod\text{-}type\}}^{m::\{mod\text{-}type\}}$
assumes r : $rank \ A = ncols \ A$
shows $A = fst \ (QR\text{-}decomposition \ A) ** snd \ (QR\text{-}decomposition \ A) \wedge$
 $pairwise \ orthogonal \ (columns \ (fst \ (QR\text{-}decomposition \ A))) \wedge$
 $(\forall i. \ norm \ (column \ i \ (fst \ (QR\text{-}decomposition \ A)))) = 1) \wedge$
 $(transpose \ (fst \ (QR\text{-}decomposition \ A))) ** (fst \ (QR\text{-}decomposition \ A)) = mat \ 1$
 \wedge
 $vec.independent \ (columns \ (fst \ (QR\text{-}decomposition \ A))) \wedge$
 $col\text{-}space \ A = col\text{-}space \ (fst \ (QR\text{-}decomposition \ A)) \wedge$

```

card (columns A) = card (columns (fst (QR-decomposition A))) ∧
invertible (snd (QR-decomposition A)) ∧
upper-triangular (snd (QR-decomposition A))
by (metis QR-decomposition-mult col-space-def full-rank-imp-is-basis2
independent-columns-fst-QR-decomposition invertible-snd-QR-decomposition
norm-columns-fst-QR-decomposition orthogonal-fst-QR-decomposition
orthogonal-matrix-fst-QR-decomposition r span-fst-QR-decomposition
upper-triangular-snd-QR-decomposition)

```

lemma *QR-decomposition-square*:

```

fixes A::realn::{mod-type}n::{mod-type}
assumes r: rank A = ncols A
shows A = fst (QR-decomposition A) ** snd (QR-decomposition A) ∧
orthogonal-matrix (fst (QR-decomposition A)) ∧
upper-triangular (snd (QR-decomposition A)) ∧
invertible (snd (QR-decomposition A)) ∧
pairwise orthogonal (columns (fst (QR-decomposition A))) ∧
(∀ i. norm (column i (fst (QR-decomposition A))) = 1) ∧
vec.independent (columns (fst (QR-decomposition A))) ∧
col-space A = col-space (fst (QR-decomposition A)) ∧
card (columns A) = card (columns (fst (QR-decomposition A)))
by (metis QR-decomposition orthogonal-matrix-fst-QR-decomposition' r)

```

QR for computing determinants

lemma *det-QR-decomposition*:

```

fixes A::realn::{mod-type}n::{mod-type}
assumes r: rank A = ncols A
shows |det A| = |(prod (λ i. snd(QR-decomposition A)$i$i) (UNIV::'n set))|
proof -
let ?Q=fst(QR-decomposition A)
let ?R=snd(QR-decomposition A)
have det-R: det ?R = (prod (λ i. snd(QR-decomposition A)$i$i) (UNIV::'n set))
  apply (rule det-upperdiagonal)
  using upper-triangular-snd-QR-decomposition[OF r]
  unfolding upper-triangular-def by simp
have |det A| = |det ?Q * det ?R| by (metis QR-decomposition-mult det-mul r)
also have ... = |det ?Q| * |det ?R| unfolding abs-mult ..
also have ... = 1 * |det ?R| using det-orthogonal-matrix[OF orthogonal-matrix-fst-QR-decomposition'[OF
r]]
  by auto
also have ... = |det ?R| by simp
also have ... = |(prod (λ i. snd(QR-decomposition A)$i$i) (UNIV::'n set))| un-
folding det-R ..
finally show ?thesis .
qed
end

```

5 Least Squares Approximation

```
theory Least-Squares-Approximation
imports
  QR-Decomposition
begin
```

5.1 Second part of the Fundamental Theorem of Linear Algebra

See http://en.wikipedia.org/wiki/Fundamental_theorem_of_linear_algebra

```
lemma null-space-orthogonal-complement-row-space:
  fixes A::real^cols^rows::{finite,wellorder}
  shows null-space A = orthogonal-complement (row-space A)
proof (unfold null-space-def orthogonal-complement-def, auto)
  fix x xa assume Ax: A *v x = 0 and xa: xa ∈ row-space A
  obtain y where y: xa = transpose A *v y using xa unfolding row-space-eq by
  blast
  have y v* A = xa
    using transpose-vector y by fastforce
  thus orthogonal x xa unfolding orthogonal-def
    using Ax dot-lmul-matrix inner-commute inner-zero-right
    by (metis Ax dot-lmul-matrix inner-commute inner-zero-right)
next
  fix x assume xa: ∀ xa∈row-space A. orthogonal x xa
  show A *v x = 0
    using xa unfolding row-space-eq orthogonal-def
    by (auto, metis transpose-transpose dot-lmul-matrix inner-eq-zero-iff trans-
  pose-vector)
qed
```

```
lemma left-null-space-orthogonal-complement-col-space:
  fixes A::real^cols::{finite,wellorder}^rows
  shows left-null-space A = orthogonal-complement (col-space A)
  using null-space-orthogonal-complement-row-space[of transpose A]
  unfolding left-null-space-eq-null-space-transpose
  unfolding col-space-eq-row-space-transpose .
```

5.2 Least Squares Approximation

See https://people.math.osu.edu/husen.1/teaching/571/least_squares.pdf

Part 3 of the Theorem 1.7 in the previous website.

```
lemma least-squares-approximation:
  fixes X::'a::{euclidean-space} set
  assumes subspace-S: subspace S
  and ind-X: independent X
  and X: X ⊆ S
```

and $\text{span-}X: S \subseteq \text{span } X$
and o : pairwise orthogonal X
and not-eq : proj-onto $v X \neq y$
and $y: y \in S$
shows $\text{norm } (v - \text{proj-onto } v X) < \text{norm } (v - y)$
proof –
have $S\text{-eq-span}X: S = \text{span } X$
using $X \text{ span-}X \text{ span-subspace } \text{subspace-}S$ **by** *auto*
let $?p = \text{proj-onto } v X$
have $\text{not-0}: (\text{norm}(?p - y))^2 \neq 0$
by (*metis* (*lifting*) *eq-iff-diff-eq-0 norm-eq-zero not-eq power-eq-0-iff*)
have $\text{norm } (v - y)^2 = \text{norm } (v - ?p + ?p - y)^2$ **by** *auto*
also have $\dots = \text{norm } ((v - ?p) + (?p - y))^2$
unfolding *add.assoc[symmetric]* **by** *simp*
also have $\dots = (\text{norm } (v - ?p))^2 + (\text{norm}(?p - y))^2$
proof (*rule pythagorean-theorem-norm, rule in-orthogonal-complement-imp-orthogonal*)

show $?p - y \in S$ **unfolding** *proj-onto-def proj-def[abs-def]*
proof (*rule subspace-diff[OF subspace-S - y],*
rule subspace-sum[OF subspace-S])
show $x \in X \implies (v \cdot x / (x \cdot x)) *_{\mathbb{R}} x \in S$ **for** x
by (*metis S-eq-spanX X rev-subsetD span-mul*)
qed
show $v - ?p \in \text{orthogonal-complement } S$
using *v-minus-p-orthogonal-complement assms* **by** *auto*
qed
finally have $\text{norm } (v - ?p)^2 < \text{norm } (v - y)^2$ **using** *not-0* **by** *fastforce*
thus $?thesis$ **by** (*metis (full-types) norm-gt-square power2-norm-eq-inner*)
qed

lemma *least-squares-approximation2*:
fixes $S::'a::\{\text{euclidean-space}\}$ *set*
assumes $\text{subspace-}S$: *subspace* S
and $y: y \in S$
shows $\exists p \in S. \text{norm } (v - p) \leq \text{norm } (v - y) \wedge (v - p) \in \text{orthogonal-complement } S$
proof –
obtain X **where** $\text{ind-}X$: *independent* X
and $X: X \subseteq S$
and $\text{span-}X: S \subseteq \text{span } X$
and o : pairwise orthogonal X
by (*metis order-refl orthonormal-basis-subspace subspace-S*)
let $?p = \text{proj-onto } v X$
show $?thesis$
proof (*rule bexI[of - ?p], rule conjI*)
show $\text{norm } (v - \text{proj-onto } v X) \leq \text{norm } (v - y)$
proof (*cases ?p=y*)
case *True* **thus** $\text{norm } (v - ?p) \leq \text{norm } (v - y)$ **by** *simp*


```

next
  case False
  have  $\text{norm } (v - ?p) < \text{norm } (v - y)$ 
    by (rule least-squares-approximation[OF subspace-S ind-X X span-X o False
y])
  thus  $\text{norm } (v - ?p) \leq \text{norm } (v - y)$  by simp
qed
show  $?p \in S$ 
  using [[unfold-abs-def = false]]
proof (unfold proj-onto-def proj-def, rule subspace-sum)
  show subspace S using subspace-S .
  show  $x \in X \implies \text{proj } v \ x \in S$  for  $x$ 
    by (simp add: proj-def X rev-subsetD subspace-S subspace-mul)
qed
show  $v - ?p \in \text{orthogonal-complement } S$ 
  by (rule v-minus-p-orthogonal-complement[OF subspace-S ind-X X span-X o])
qed
qed

corollary least-squares-approximation3:
  fixes  $S::'a::\{\text{euclidean-space}\}$  set
  assumes subspace-S: subspace S
  shows  $\exists p \in S. \forall y \in S. \text{norm } (v - p) \leq \text{norm } (v - y) \wedge (v - p) \in \text{orthogonal-complement } S$ 
proof –
  obtain  $X$  where ind-X: independent X
  and  $X: X \subseteq S$ 
  and span-X: S ⊆ span X
  and  $o$ : pairwise orthogonal X
  by (metis order-refl orthonormal-basis-subspace subspace-S)
  let  $?p = \text{proj-onto } v \ X$ 
  show ?thesis
  proof (rule bestI[of - ?p], auto)
    fix  $y$  assume  $y: y \in S$ 
    show  $\text{norm } (v - ?p) \leq \text{norm } (v - y)$ 
    proof (cases  $?p = y$ )
      case True thus ?thesis by simp
    next
      case False
      have  $\text{norm } (v - ?p) < \text{norm } (v - y)$ 
        by (rule least-squares-approximation[OF subspace-S ind-X X span-X o False
y])
      thus ?thesis by simp
    qed
  show  $v - ?p \in \text{orthogonal-complement } S$ 
    by (rule v-minus-p-orthogonal-complement[OF subspace-S ind-X X span-X o])
  next
  show  $?p \in S$ 
  proof (unfold proj-onto-def, rule subspace-sum)

```

show *subspace S using subspace-S* .
show $x \in X \implies \text{proj } v \ x \in S$ **for** x
by (*metis Projections.proj-def X subset-iff subspace-S subspace-mul*)
qed
qed
qed

lemma *norm-least-squares*:

fixes $A::\text{real}^{\wedge}\text{cols}::\{\text{finite,wellorder}\}^{\wedge}\text{rows}$
shows $\exists x. \forall x'. \text{norm } (b - A *v x) \leq \text{norm } (b - A *v x')$
proof –
have $\exists p \in \text{col-space } A. \forall y \in \text{col-space } A. \text{norm } (b - p) \leq \text{norm } (b - y) \wedge (b-p) \in \text{orthogonal-complement } (\text{col-space } A)$
using *least-squares-approximation3[OF subspace-col-space[of A, unfolded subspace-vec-eq]]* .
from this obtain p **where** $p: p \in \text{col-space } A$ **and** *least*: $\forall y \in \text{col-space } A. \text{norm } (b - p) \leq \text{norm } (b - y)$
and *bp-orthogonal*: $(b-p) \in \text{orthogonal-complement } (\text{col-space } A)$
by *blast*
obtain x **where** $x: p = A *v x$ **using** p **unfolding** *col-space-eq* **by** *blast*
show *?thesis*
proof (*rule exI[of - x], auto*)
fix x'
have $A *v x' \in \text{col-space } A$ **unfolding** *col-space-eq* **by** *auto*
thus $\text{norm } (b - A *v x) \leq \text{norm } (b - A *v x')$ **using** *least* **unfolding** x **by** *auto*
qed
qed

definition *set-least-squares-approximation* $A \ b = \{x. \forall y. \text{norm } (b - A *v x) \leq \text{norm } (b - A *v y)\}$

corollary *least-squares-approximation4*:

fixes $S::'a::\{\text{euclidean-space}\}$ *set*
assumes *subspace-S*: *subspace S*
shows $\exists! p \in S. \forall y \in S - \{p\}. \text{norm } (v - p) < \text{norm } (v - y)$
proof (*auto*)
obtain X **where** *ind-X*: *independent X*
and $X: X \subseteq S$
and *span-X*: $S \subseteq \text{span } X$
and o : *pairwise orthogonal X*
by (*metis order-refl orthonormal-basis-subspace subspace-S*)
let $?p = \text{sum } (\text{proj } v) \ X$
show $\exists p. p \in S \wedge (\forall y \in S - \{p\}. \text{norm } (v - p) < \text{norm } (v - y))$
proof (*rule exI[of - ?p], rule conjI, rule subspace-sum*)
show *subspace S using subspace-S* .
show $x \in X \implies \text{proj } v \ x \in S$ **for** x
by (*metis Projections.proj-def X subset-iff subspace-S subspace-mul*)
show $\forall y \in S - \{?p\}. \text{norm } (v - ?p) < \text{norm } (v - y)$

using X *ind-X least-squares-approximation* o *span-X subspace-S proj-onto-def*
by (*metis (mono-tags) Diff-iff singletonI*)
qed
fix $p y$
assume $p: p \in S$
and $\forall y \in S - \{p\}. \text{norm } (v - p) < \text{norm } (v - y)$
and $y \in S$
and $\forall ya \in S - \{y\}. \text{norm } (v - y) < \text{norm } (v - ya)$
thus $p = y$ **by** (*metis member-remove not-less-iff-gr-or-eq remove-def*)
qed

corollary *least-squares-approximation4'*:
fixes $S::'a::\{\text{euclidean-space}\}$ *set*
assumes *subspace-S: subspace S*
shows $\exists! p \in S. \forall y \in S. \text{norm } (v - p) \leq \text{norm } (v - y)$
proof (*auto*)
obtain X **where** *ind-X: independent X*
and $X: X \subseteq S$
and *span-X: S ⊆ span X*
and $o: \text{pairwise orthogonal } X$
by (*metis order-refl orthonormal-basis-subspace subspace-S*)
let $?p = \text{sum } (\text{proj } v) X$
show $\exists p. p \in S \wedge (\forall y \in S. \text{norm } (v - p) \leq \text{norm } (v - y))$
proof (*rule exI[of - ?p], rule conjI, rule subspace-sum*)
show *subspace S using subspace-S .*
show $x \in X \implies \text{proj } v x \in S$ **for** x
by (*metis Projections.proj-def X subset-iff subspace-S subspace-mul*)
show $\forall y \in S. \text{norm } (v - ?p) \leq \text{norm } (v - y)$
by (*metis (mono-tags) proj-onto-def X dual-order.refl ind-X*
least-squares-approximation less-imp-le o span-X subspace-S)
qed
fix $p y$
assume $p: p \in S$ **and** $p': \forall y \in S. \text{norm } (v - p) \leq \text{norm } (v - y)$
and $y: y \in S$ **and** $y': \forall ya \in S. \text{norm } (v - y) \leq \text{norm } (v - ya)$
obtain a **where** $a: a \in S$ **and** $a': \forall y \in S - \{a\}. \text{norm } (v - a) < \text{norm } (v - y)$
and $a\text{-uniq}: \forall b. (b \in S \wedge (\forall c \in S - \{b\}. \text{norm } (v - b) < \text{norm } (v - c))) \implies b = a$
using *least-squares-approximation4 [OF subspace-S]*
by *metis*
have $p = a$ **using** $p p' a\text{-uniq}$ *leD* **by** (*metis a a' member-remove remove-def*)
moreover **have** $y = a$ **using** $y y' a\text{-uniq}$
by (*metis a a' leD member-remove remove-def*)
ultimately show $p = y$ **by** *simp*
qed

corollary *least-squares-approximation5*:
fixes $S::'a::\{\text{euclidean-space}\}$ *set*
assumes *subspace-S: subspace S*
shows $\exists! p \in S. \forall y \in S - \{p\}. \text{norm } (v - p) < \text{norm } (v - y) \wedge v - p \in \text{orthogo-}$

nal-complement S
proof (*auto*)
obtain X **where** $ind-X$: *independent X*
and X : $X \subseteq S$
and $span-X$: $S \subseteq span X$
and o : *pairwise orthogonal X*
by (*metis order-refl orthonormal-basis-subspace subspace-S*)
let $?p=sum (proj v) X$
show $\exists p. p \in S \wedge (\forall y \in S - \{p\}. norm (v - p) < norm (v - y) \wedge v - p \in$
orthogonal-complement S)
proof (*rule exI[of - ?p], rule conjI, rule subspace-sum*)
show *subspace S using subspace-S .*
show $x \in X \implies proj v x \in S$ **for** x
by (*simp add: Projections.proj-def X rev-subsetD subspace-S subspace-mul*)
have $\forall y \in S - \{?p\}. norm (v - ?p) < norm (v - y)$
using *least-squares-approximation[OF subspace-S ind-X X span-X o]*
unfolding *proj-onto-def*
by (*metis (no-types) member-remove remove-def*)
moreover have $v - ?p \in orthogonal-complement S$
by (*metis (no-types) X ind-X o span-X subspace-S v-minus-p-orthogonal-complement*
proj-onto-def)
ultimately show $\forall y \in S - \{?p\}. norm (v - ?p) < norm (v - y) \wedge v - ?p \in$
orthogonal-complement S
by *auto*
qed
fix $p y$
assume $p: p \in S$ **and** $p': \forall y \in S - \{p\}. norm (v - p) < norm (v - y) \wedge v - p$
 $\in orthogonal-complement S$
and $y: y \in S$ **and** $y': \forall ya \in S - \{y\}. norm (v - y) < norm (v - ya) \wedge v - y$
 $\in orthogonal-complement S$
show $p=y$
by (*metis least-squares-approximation4 p p' subspace-S y y'*)
qed

corollary *least-squares-approximation5'*:
fixes $S::'a::\{euclidean-space\}$ *set*
assumes *subspace-S: subspace S*
shows $\exists! p \in S. \forall y \in S. norm (v - p) \leq norm (v - y) \wedge v - p \in orthogonal-complement$
 S
by (*metis least-squares-approximation3 least-squares-approximation4' subspace-S*)

corollary *least-squares-approximation6*:
fixes $S::'a::\{euclidean-space\}$ *set*
assumes *subspace-S: subspace S*
and $p \in S$
and $\forall y \in S. norm (v - p) \leq norm (v - y)$
shows $v - p \in orthogonal-complement S$
proof –
obtain a **where** $a: a \in S$ **and** $a': \forall y \in S. norm (v - a) \leq norm (v - y) \wedge v - a$

\in *orthogonal-complement S*
and $\forall b. (b \in S \wedge (\forall y \in S. \text{norm } (v - b) \leq \text{norm } (v - y) \wedge v - b \in \text{orthogonal-complement } S)) \longrightarrow b = a$
using *least-squares-approximation5'* [*OF subspace-S*] **by** *metis*
have $p = a$
by (*metis a' assms(2) assms(3) least-squares-approximation4'* *subspace-S*)
thus *?thesis using a'* **by** (*metis assms(2)*)
qed

corollary *least-squares-approximation7*:
fixes $S :: 'a :: \{\text{euclidean-space}\}$ *set*
assumes *subspace-S: subspace S*
and $v - p \in \text{orthogonal-complement } S$
and $p \in S$
and $y \in S$
shows $\text{norm } (v - p) \leq \text{norm } (v - y)$
proof (*cases y=p*)
case *True thus ?thesis by simp*
next
case *False*
have $\text{norm } (v - y)^2 = \text{norm } ((v - p) + (p - y))^2$
by (*metis (opaque-lifting, no-types) add-diff-cancel-left add-ac(1) add-diff-add add-diff-cancel*)
also have $\dots = \text{norm } (v - p)^2 + \text{norm } (p - y)^2$
proof (*rule pythagorean-theorem-norm, rule in-orthogonal-complement-imp-orthogonal*)
show $p - y \in S$ **by** (*metis assms(3) assms(4) subspace-S subspace-diff*)
show $v - p \in \text{orthogonal-complement } S$ **by** (*metis assms(2)*)
qed
finally have $\text{norm } (v - p)^2 \leq \text{norm } (v - y)^2$ **by** *auto*
thus $\text{norm } (v - p) \leq \text{norm } (v - y)$ **by** (*metis norm-ge-zero power2-le-imp-le*)
qed

lemma *in-set-least-squares-approximation*:
fixes $A :: \text{real}^{\text{cols}} :: \{\text{finite, wellorder}\}^{\text{rows}}$
assumes $o: A * v \ x - b \in \text{orthogonal-complement } (\text{col-space } A)$
shows $(x \in \text{set-least-squares-approximation } A \ b)$
proof (*unfold set-least-squares-approximation-def, auto*)
fix y
show $\text{norm } (b - A * v \ x) \leq \text{norm } (b - A * v \ y)$
proof (*rule least-squares-approximation7*)
show *subspace (col-space A) using subspace-col-space[of A, unfolded subspace-vec-eq]* .
show $b - A * v \ x \in \text{orthogonal-complement } (\text{col-space } A)$
using *o subspace-orthogonal-complement[of (col-space A)]*
using *minus-diff-eq subspace-neg* **by** *metis*
show $A * v \ x \in \text{col-space } A$ **unfolding** *col-space-eq[of A]* **by** *auto*
show $A * v \ y \in \text{col-space } A$ **unfolding** *col-space-eq* **by** *auto*

qed
qed

lemma *in-set-least-squares-approximation-eq*:

fixes $A::\text{real}^{\wedge}\text{cols}::\{\text{finite},\text{wellorder}\}^{\wedge}\text{rows}$

shows $(x \in \text{set-least-squares-approximation } A \ b) = (\text{transpose } A \ ** \ A \ *v \ x = \text{transpose } A \ *v \ b)$

proof

assume $x: x \in \text{set-least-squares-approximation } A \ b$

hence $a: \forall a. \text{norm } (b - A \ *v \ x) \leq \text{norm } (b - A \ *v \ a)$ **unfolding** *set-least-squares-approximation-def* **by** *simp*

have $b - A \ *v \ x \in \text{orthogonal-complement } (\text{col-space } A)$

proof (*rule least-squares-approximation6*)

show *subspace* $(\text{col-space } A)$ **using** *subspace-col-space*[*of A, unfolded subspace-vec-eq*].

show $A \ *v \ x \in \text{col-space } A$ **unfolding** *col-space-eq*[*of A*] **by** *auto*

show $\forall y \in \text{col-space } A. \text{norm } (b - A \ *v \ x) \leq \text{norm } (b - y)$ **using** *a* **unfolding** *col-space-eq* **by** *auto*

qed

hence $b - A \ *v \ x \in \text{null-space } (\text{transpose } A)$

unfolding *null-space-orthogonal-complement-row-space*

unfolding *col-space-eq-row-space-transpose*.

hence $\text{transpose } A \ *v \ (b - A \ *v \ x) = 0$ **unfolding** *null-space-def* **by** *simp*

thus $(\text{transpose } A \ ** \ A) \ *v \ x = (\text{transpose } A) \ *v \ b$

by (*metis eq-iff-diff-eq-0 matrix-vector-mul-assoc matrix-vector-right-distrib-minus*)

next

assume $\text{transpose } A \ ** \ A \ *v \ x = \text{transpose } A \ *v \ b$

hence $(\text{transpose } A) \ *v \ (A \ *v \ x - b) = 0$

by (*metis diff-self matrix-vector-mul-assoc matrix-vector-right-distrib-minus*)

hence $(A \ *v \ x - b) \in \text{null-space } (\text{transpose } A)$ **unfolding** *null-space-def* **by** *simp*

hence $(A \ *v \ x - b) \in \text{orthogonal-complement } (\text{col-space } A)$

by (*metis left-null-space-eq-null-space-transpose left-null-space-orthogonal-complement-col-space*)

thus $x \in \text{set-least-squares-approximation } A \ b$ **by** (*rule in-set-least-squares-approximation*)

qed

lemma *in-set-least-squares-approximation-eq-full-rank*:

fixes $A::\text{real}^{\wedge}\text{cols}::\text{mod-type}^{\wedge}\text{rows}::\text{mod-type}$

assumes $r: \text{rank } A = \text{ncols } A$

shows $(x \in \text{set-least-squares-approximation } A \ b) = (x = \text{matrix-inv } (\text{transpose } A \ ** \ A) \ ** \ \text{transpose } A \ *v \ b)$

proof –

have *int-tA*: *invertible* $(\text{transpose } A \ ** \ A)$ **using** *invertible-transpose-mult*[*OF r*]

.

show *?thesis*

proof

fix x **assume** $x \in \text{set-least-squares-approximation } A \ b$

hence $\text{transpose } A \ ** \ A \ *v \ x = \text{transpose } A \ *v \ b$ **using** *in-set-least-squares-approximation-eq*

```

by auto
  thus  $x = \text{matrix-inv } (\text{transpose } A ** A) ** \text{transpose } A * v b$ 
  by (metis int-tA matrix-inv-left matrix-vector-mul-assoc matrix-vector-mul-lid)
next
  fix  $x$  assume  $x = \text{matrix-inv } (\text{transpose } A ** A) ** \text{transpose } A * v b$ 
  hence  $\text{transpose } A ** A * v x = \text{transpose } A * v b$ 
  by (metis int-tA matrix-inv-right matrix-vector-mul-assoc matrix-vector-mul-lid)
  thus  $x \in \text{set-least-squares-approximation } A b$  unfolding in-set-least-squares-approximation-eq
.
qed
qed

```

lemma *in-set-least-squares-approximation-eq-full-rank-QR*:

```

fixes  $A::\text{real}^{\wedge}\text{cols}::\{\text{mod-type}\}^{\wedge}\text{rows}::\{\text{mod-type}\}$ 
assumes  $r: \text{rank } A = \text{ncols } A$ 
shows  $(x \in \text{set-least-squares-approximation } A b) = ((\text{snd } (\text{QR-decomposition } A)) * v x = \text{transpose } (\text{fst } (\text{QR-decomposition } A)) * v b)$ 
proof -
  let  $?Q = \text{fst } (\text{QR-decomposition } A)$ 
  let  $?R = \text{snd } (\text{QR-decomposition } A)$ 
  have  $\text{inv-tR}: \text{invertible } (\text{transpose } ?R)$ 
  by (metis invertible-snd-QR-decomposition invertible-transpose r)
  have  $\text{inv-inv-tR}: \text{invertible } (\text{matrix-inv } (\text{transpose } ?R))$ 
  by (metis inv-tR invertible-fst-Gauss-Jordan-PA matrix-inv-Gauss-Jordan-PA)
  have  $(x \in \text{set-least-squares-approximation } A b) = (\text{transpose } A ** A * v x = \text{transpose } A * v b)$ 
  using in-set-least-squares-approximation-eq .
  also have  $\dots = (\text{transpose } (?Q ** ?R) ** (?Q ** ?R) * v x = \text{transpose } (?Q ** ?R) * v b)$ 
  using QR-decomposition-mult[OF r] by simp
  also have  $\dots = (\text{transpose } ?R ** \text{transpose } ?Q ** (?Q ** ?R) * v x = \text{transpose } ?R ** \text{transpose } ?Q * v b)$ 
  by (metis (opaque-lifting, no-types) matrix-transpose-mul)
  also have  $\dots = (\text{transpose } ?R * v (\text{transpose } ?Q ** (?Q ** ?R) * v x) = \text{transpose } ?R * v (\text{transpose } ?Q * v b))$ 
  by (metis (opaque-lifting, no-types) matrix-vector-mul-assoc)
  also have  $\dots = (\text{matrix-inv } (\text{transpose } ?R) * v (\text{transpose } ?R * v (\text{transpose } ?Q * v b)) * v x)$ 
  =  $\text{matrix-inv } (\text{transpose } ?R) * v (\text{transpose } ?R * v (\text{transpose } ?Q * v b))$ 
  using inv-matrix-vector-mul-left[OF inv-inv-tR] by auto
  also have  $\dots = ((\text{matrix-inv } (\text{transpose } ?R) ** \text{transpose } ?R) * v (\text{transpose } ?Q * v b)) * v x$ 
  =  $(\text{matrix-inv } (\text{transpose } ?R) ** \text{transpose } ?R) * v (\text{transpose } ?Q * v b)$ 
  by (metis (opaque-lifting, no-types) matrix-vector-mul-assoc)
  also have  $\dots = (\text{transpose } ?Q ** (?Q ** ?R) * v x = \text{transpose } ?Q * v b)$ 
  unfolding matrix-inv-left[OF inv-tR]
  unfolding matrix-vector-mul-lid ..

```

also have ... = ((*transpose* ?Q ** ?Q) ** ?R *v x = *transpose* ?Q *v b)
by (*metis* (*opaque-lifting*, *no-types*) *matrix-mul-assoc*)
also have ... = (?R *v x = *transpose* ?Q *v b)
unfolding *orthogonal-matrix-fst-QR-decomposition*[OF r]
unfolding *matrix-mul-lid* ..
finally show (x ∈ *set-least-squares-approximation* A b) = (?R *v x = (*transpose* ?Q) *v b) .
qed

corollary *in-set-least-squares-approximation-eq-full-rank-QR2*:

fixes A::*real*^*cols*::{*mod-type*}^*rows*::{*mod-type*}
assumes r: *rank* A = *ncols* A
shows (x ∈ *set-least-squares-approximation* A b) = (x = *matrix-inv* (*snd* (*QR-decomposition* A)) ** *transpose* (*fst* (*QR-decomposition* A)) *v b)
proof –
let ?Q = *fst* (*QR-decomposition* A)
let ?R = *snd* (*QR-decomposition* A)
have *inv-R*: *invertible* ?R **by** (*metis* *invertible-snd-QR-decomposition* r)
have (x ∈ *set-least-squares-approximation* A b) = (?R *v x = *transpose* ?Q *v b)
using *in-set-least-squares-approximation-eq-full-rank-QR*[OF r] .
also have ... = (*matrix-inv* ?R ** ?R *v x = *matrix-inv* ?R ** *transpose* ?Q *v b)
by (*metis* (*opaque-lifting*, *no-types*) *Gauss-Jordan-PA-eq calculation fst-Gauss-Jordan-PA inv-R inv-matrix-vector-mul-left invertible-fst-Gauss-Jordan-PA matrix-inv-Gauss matrix-vector-mul-assoc*)
also have ... = (x = *matrix-inv* ?R ** *transpose* ?Q *v b)
by (*metis* *inv-R matrix-inv-left matrix-vector-mul-lid*)
finally show (x ∈ *set-least-squares-approximation* A b) = (x = *matrix-inv* ?R ** *transpose* ?Q *v b) .
qed

lemma *set-least-squares-approximation-unique-solution*:

fixes A::*real*^*cols*::{*mod-type*}^*rows*::{*mod-type*}
assumes r: *rank* A = *ncols* A
shows (*set-least-squares-approximation* A b) = {*matrix-inv* (*transpose* A ** A)***transpose* A *v b}
by (*metis* (*opaque-lifting*, *mono-tags*) *empty-iff in-set-least-squares-approximation-eq-full-rank empty-iff insertI1 r subsetI subset-singletonD*)

lemma *set-least-squares-approximation-unique-solution-QR*:

fixes A::*real*^*cols*::{*mod-type*}^*rows*::{*mod-type*}
assumes r: *rank* A = *ncols* A
shows (*set-least-squares-approximation* A b) = {*matrix-inv* (*snd* (*QR-decomposition* A)) ** *transpose* (*fst* (*QR-decomposition* A)) *v b}
by (*metis* (*opaque-lifting*, *mono-tags*) *empty-iff in-set-least-squares-approximation-eq-full-rank-QR2 insertI1 r subsetI subset-singletonD*)

end

6 Examples of execution using floats

```
theory Examples-QR-Abstract-Float
imports
  QR-Decomposition
  Gauss-Jordan.Code-Real-Approx-By-Float-Haskell
begin
```

6.0.1 Examples

```
definition example1 = (let A = list-of-list-to-matrix [[1,2,4],[9,4,5],[0,0,0]]::real^3^3
in
  matrix-to-list-of-list (divide-by-norm A))
```

```
definition example2 = (let A = list-of-list-to-matrix [[1,2,4],[9,4,5],[0,0,4]]::real^3^3
in
  matrix-to-list-of-list (fst (QR-decomposition A)))
```

```
definition example3 = (let A = list-of-list-to-matrix [[1,2,4],[9,4,5],[0,0,4]]::real^3^3
in
  matrix-to-list-of-list (snd (QR-decomposition A)))
```

```
definition example4 = (let A = list-of-list-to-matrix [[1,2,4],[9,4,5],[0,0,4]]::real^3^3
in
  matrix-to-list-of-list (fst (QR-decomposition A) ** (snd (QR-decomposition A))))
```

```
definition example5 = (let A = list-of-list-to-matrix [[1,sqrt 2,4],[sqrt 5,4,5],[0,sqrt
7,4]]::real^3^3 in
  matrix-to-list-of-list (fst (QR-decomposition A)))
```

```
export-code example1 example2 example3 example4 example5 in SML mod-
ule-name QR
```

end

7 Examples of execution using symbolic computation

```
theory Examples-QR-Abstract-Symbolic
imports
  QR-Decomposition
  Real-Impl.Real-Unique-Impl
begin
```

7.1 Execution of the QR decomposition using symbolic computation

7.1.1 Some previous definitions and lemmas

The symbolic computation is based on the René Thiemann's work about implementing field extensions of the form $\mathbb{Q}[\sqrt{b}]$.

definition *show-vec-real* $v = (\chi \ i. \text{show-real } (v \ \$ \ i))$

lemma *[code abstract]*: $\text{vec-nth } (\text{show-vec-real } v) = (\% \ i. \text{show-real } (v \ \$ \ i))$

unfolding *show-vec-real-def* **by** *auto*

definition *show-matrix-real* $A = (\chi \ i. \text{show-vec-real } (A \ \$ \ i))$

lemma*[code abstract]*: $\text{vec-nth } (\text{show-matrix-real } A) = (\% \ i. \text{show-vec-real } (A \ \$ \ i))$

unfolding *show-matrix-real-def* **by** *auto*

7.1.2 Examples

value *let* $A = \text{list-of-list-to-matrix } [[1,2,4],[9,4,5],[0,0,0]]::\text{real}^3^3$ *in*
matrix-to-list-of-list (*show-matrix-real* (*divide-by-norm* A))

value *let* $A = \text{list-of-list-to-matrix } [[1,2,4],[9,4,5],[0,0,4]]::\text{real}^3^3$ *in*
matrix-to-list-of-list (*show-matrix-real* (*fst* (*QR-decomposition* A))))

value *let* $A = \text{list-of-list-to-matrix } [[1,2,4],[9,4,5],[0,0,4]]::\text{real}^3^3$ *in*
matrix-to-list-of-list (*show-matrix-real* (*snd* (*QR-decomposition* A))))

value *let* $A = \text{list-of-list-to-matrix } [[1,2,4],[9,4,5],[0,0,4]]::\text{real}^3^3$ *in*
matrix-to-list-of-list (*show-matrix-real* ((*fst* (*QR-decomposition* A)) ** (*snd* (*QR-decomposition* A))))))

value *let* $A = \text{list-of-list-to-matrix } [[1,2,4],[9,4,5],[0,0,4],[3,5,4]]::\text{real}^3^4$ *in*
matrix-to-list-of-list (*show-matrix-real* ((*fst* (*QR-decomposition* A)) ** (*snd* (*QR-decomposition* A))))))

value *let* $A = \text{list-of-list-to-matrix } [[1,2,1],[9,4,9],[2,0,2],[0,5,0]]::\text{real}^3^4$ *in*
matrix-to-list-of-list (*show-matrix-real* ((*fst* (*QR-decomposition* A)) ** (*snd* (*QR-decomposition* A))))))

value *let* $A = \text{list-of-list-to-matrix } [[1,2,1],[9,4,9],[2,0,2],[0,5,0]]::\text{real}^3^4$ *in*
matrix-to-list-of-list (*show-matrix-real* (*fst* (*QR-decomposition* A))))

value *let* $A = \text{list-of-list-to-matrix } [[1,2,1],[9,4,9],[2,0,2],[0,5,0]]::\text{real}^3^4$ *in*
vec-to-list (*show-vec-real* ((*column* 0 (*fst* (*QR-decomposition* A))))))

value *let* $A = \text{list-of-list-to-matrix } [[1,2,1],[9,4,9],[2,0,2],[0,5,0]]::\text{real}^3^4$ *in*

```

vec-to-list (show-vec-real ((column 1 (fst (QR-decomposition A))))))

value let A = list-of-list-to-matrix [[1,2,1],[9,4,9],[2,0,2],[0,5,0]]::real^3^4 in
matrix-to-list-of-list (show-matrix-real (snd (QR-decomposition A)))

value let A = list-of-list-to-matrix [[1,2,1],[9,4,9]]::real^3^2 in
matrix-to-list-of-list (show-matrix-real ((fst (QR-decomposition A)) ** (snd (QR-decomposition
A))))

value let A = list-of-list-to-matrix [[1,2,1],[9,4,9]]::real^3^2 in
matrix-to-list-of-list (show-matrix-real ((fst (QR-decomposition A))))

value let A = list-of-list-to-matrix [[1,2,1],[9,4,9]]::real^3^2 in
matrix-to-list-of-list (show-matrix-real ((snd (QR-decomposition A))))

definition example1 = (let A = list-of-list-to-matrix [[1,2,1],[9,4,9]]::real^3^2 in
matrix-to-list-of-list (show-matrix-real ((snd (QR-decomposition A))))))

export-code example1 in SML module-name QR

end

```

8 IArray Addenda QR

```

theory IArray-Addenda-QR
imports
  HOL-Library.IArray
begin

```

The new file about Iarrays, with different instantiations from the presented ones in the Gauss-Jordan algorithm.

In order to make the formalisation of the QR algorithm easier, we have decided to present here some alternative instantiations for immutable arrays. Let see an example. The following definition is the one presented in the Gauss-Jordan AFP entry to sum two vectors:

```
plus-iarray A B = IArray.of-fun ( $\lambda n. A !! n + B !! n$ ) (IArray.length A)
```

While the following is the one we will present in this development:

```

plus-iarray A B =
  (let length-A = (IArray.length A);
    length-B = (IArray.length B);

```

```

n=max length-A length-B ;
A'= IArray.of-fun (λa. if a < length-A then A!!a else 0) n;
B'=IArray.of-fun (λa. if a < length-B then B!!a else 0) n
in IArray.of-fun (λa. A' !! a + B' !! a) n)

```

Now the sum is done up to the length of the shortest vector and it is completed with zeros up to the length of the largest vector. This allows us to prove that `iarray` is an instance of `comm-monoid-add`, which is quite useful for the QR algorithm (we will be able to do sums involving immutable arrays).

These are just alternative definitions of the main operations over immutable arrays. They have the advantage of being an instance of `comm-monoid-add`; nevertheless, the performance is slower and proofs become more cumbersome. The user should decide what definitions to use (the presented here or the presented ones in the Gauss-Jordan AFP entry) depending on the algorithm to formalise.

```

lemma iarray-exhaust2:
  (xs = ys) = (IArray.list-of xs = IArray.list-of ys)
  by (metis iarray.exhaust list-of.simps)

```

```

lemma of-fun-nth:
  assumes i: i<n
  shows (IArray.of-fun f n) !! i = f i
  unfolding IArray.of-fun-def using map-nth i by auto

```

8.1 Some previous instances

```

instantiation iarray :: (plus,zero) plus
begin

```

```

definition plus-iarray :: 'a iarray ⇒ 'a iarray ⇒ 'a iarray
  where plus-iarray A B =
    (let length-A = (IArray.length A);
        length-B = (IArray.length B);
        n=max length-A length-B ;
        A'= IArray.of-fun (λa. if a < length-A then A!!a else 0) n;
        B'=IArray.of-fun (λa. if a < length-B then B!!a else 0) n
        in
        IArray.of-fun (λa. A' !! a + B' !! a) n)

```

```

instance proof qed
end

```

```

instantiation iarray :: (zero) zero
begin
definition zero-iarray = (IArray[]::'a iarray)
instance proof qed

```

```

end

instantiation iarray :: (comm-monoid-add) comm-monoid-add
begin

instance
proof
  fix a b c::'a iarray
  have max-eq: (max (IArray.length 0) (IArray.length a)) =(IArray.length a)
  proof -
    have max (length (IArray.list-of (0::'a iarray))) (length (IArray.list-of a)) =
length (IArray.list-of a)
    by (metis list.size(3) list-of.simps max-0L zero-iarray-def)
    thus max (IArray.length 0) (IArray.length a) = IArray.length a
    by (metis IArray.length-def list.size(3) list-of.simps zero-iarray-def)
  qed
  have length0: IArray.length 0 = 0 unfolding zero-iarray-def by auto
  show 0 + a = a
  proof (unfold iarray-exhaust2 list-eq-iff-nth-eq, auto, unfold IArray.length-def[symmetric]
IArray.sub-def[symmetric])
    show length-eq: IArray.length (0 + a) = IArray.length a unfolding plus-iarray-def
Let-def using max-eq by auto
    fix i assume i: i < IArray.length (0 + a)
    have i2: i < IArray.length a by (metis length-eq i)
    have (0 + a) !! i = (λaa. IArray.of-fun (λa. if a < 0 then 0 !! a else 0)
(IArray.length a) !! aa +
IArray.of-fun (λaa. if aa < IArray.length a then a !! aa else 0) (IArray.length
a) !! aa) i
    unfolding plus-iarray-def Let-def
    unfolding max-eq unfolding length0 unfolding sub-def[symmetric]
by (rule of-fun-nth[OF i2])
    also have ... = (λaa. 0 + IArray.of-fun (λaa. if aa < IArray.length a then a
!! aa else 0) (IArray.length a) !! aa) i
    using i2 by auto
    also have ... = a !! i using i2 by simp
    finally show (0 + a) !! i = a !! i .
  qed
  show a + b = b + a
  proof (unfold iarray-exhaust2 list-eq-iff-nth-eq, auto, unfold IArray.length-def[symmetric]
IArray.sub-def[symmetric])
    have max-eq: (max (IArray.length a) (IArray.length b)) = IArray.length (a +
b)
    unfolding plus-iarray-def Let-def by auto
    show length-eq: IArray.length (a + b) = IArray.length (b + a)
    unfolding plus-iarray-def Let-def by auto
    fix i assume i: i < IArray.length (a + b)
    have i2: i < IArray.length (b+a) using i unfolding length-eq .
    have i3: i < (max (IArray.length a) (IArray.length b)) by (metis i max-eq)
    have i4: i < (max (IArray.length b) (IArray.length a)) by (metis i3 max commute)

```

```

show (a + b) !! i = (b + a) !! i
  unfolding plus-iarray-def Let-def
  unfolding of-fun-nth[OF i3]
  unfolding of-fun-nth[OF i4]
  by (simp only: add.commute)
qed
show a + b + c = a + (b + c)
proof (unfold iarray-exhaust2 list-eq-iff-nth-eq, auto, unfold IArray.length-def[symmetric]
IArray.sub-def[symmetric])
  show length-eq: IArray.length (a + b + c) = IArray.length (a + (b + c))
    unfolding plus-iarray-def Let-def by auto
  fix i assume i: i < IArray.length (a + b + c)
  have i2: i < (max (IArray.length (IArray.of-fun (λaa. IArray.of-fun
    (λaa. if aa < IArray.length a then a !! aa else 0) (max (IArray.length a)
(IArray.length b)) !! aa +
    IArray.of-fun (λa. if a < IArray.length b then b !! a else 0) (max (IArray.length
a) (IArray.length b)) !! aa)
    (max (IArray.length a) (IArray.length b))))
    (IArray.length c)) using i unfolding plus-iarray-def Let-def by auto
  have i3: i < (max (IArray.length a) (IArray.length (IArray.of-fun
    (λa. IArray.of-fun (λa. if a < IArray.length b then b !! a else 0) (max
(IArray.length b) (IArray.length c)) !! a +
    IArray.of-fun (λa. if a < IArray.length c then c !! a else 0) (max (IArray.length
b) (IArray.length c)) !! a)
    (max (IArray.length b) (IArray.length c))))))
    using i unfolding plus-iarray-def Let-def by auto
  show (a + b + c) !! i = (a + (b + c)) !! i
    unfolding plus-iarray-def unfolding Let-def
    unfolding of-fun-nth[OF i2]
    unfolding of-fun-nth[OF i3]
    using i2 i3
    by (auto simp add: add.assoc)
qed
qed
end

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

instantiation iarray :: ({minus,zero}) minus
begin

definition minus-iarray :: 'a iarray ⇒ 'a iarray ⇒ 'a iarray
  where minus-iarray A B =
    (let length-A = (IArray.length A);

```

```

length-B= (IArray.length B);
n=max length-A length-B ;
A'= IArray.of-fun (λa. if a < length-A then A!!a else 0) n;
B'=IArray.of-fun (λa. if a < length-B then B!!a else 0) n
in
IArray.of-fun (λa. A' !! a - B' !! a) n)

```

```

instance proof qed
end

```

8.2 Some previous definitions and properties for IArrays

8.2.1 Lemmas

8.2.2 Definitions

```

fun all :: ('a ⇒ bool) ⇒ 'a iarray ⇒ bool
  where all p (IArray as) = (ALL a : set as. p a)
hide-const (open) all

```

```

fun exists :: ('a ⇒ bool) ⇒ 'a iarray ⇒ bool
  where exists p (IArray as) = (EX a : set as. p a)
hide-const (open) exists

```

8.3 Code generation

code-printing

```

constant IArray-Addenda-QR.exists → (SML) Vector.exists
| constant IArray-Addenda-QR.all → (SML) Vector.all

```

```

end

```

9 Matrices as nested IArrays

```

theory Matrix-To-IArray-QR

```

```

imports

```

```

  Rank-Nullity-Theorem.Mod-Type
  Gauss-Jordan.Elementary-Operations
  IArray-Addenda-QR

```

```

begin

```

The file is similar to the *Matrix-To-IArray.thy* one, presented in the Gauss-Jordan algorithm. But now, some proofs have changed slightly because of the new instantiations presented in the file *IArray-Addenda-QR.thy*.

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

9.1.1 Isomorphism between vec and iarray

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

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

lemma *vec-to-iarray-nth*:
fixes A::'aⁿ::{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::'aⁿ::{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::'aⁿ::{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::'aⁿ::{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 $\text{map } (\lambda i. A \$ \text{from-nat } i) [0..< \text{CARD}('n)] ! \text{to-nat } i = A \$ i$ by *simp*
 qed

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

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

9.1.2 Isomorphism between matrix and nested iarrays

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

definition *iarray-to-matrix* :: $'a \text{ iarray iarray} \Rightarrow 'a \wedge 'n::\{\text{mod-type}\} \wedge 'm::\{\text{mod-type}\}$
 where $\text{iarray-to-matrix } A = (\chi \ i \ j. A !! (\text{to-nat } i) !! (\text{to-nat } j))$

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

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

```

9.2 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 (λn. 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 (λm. A !! m !! k) (IArray.length A)

```

```

definition nrows-iarray :: 'a iarray iarray => nat

```

where $nrows\text{-}iarray\ A = IArray.length\ A$

definition $ncols\text{-}iarray :: 'a\ iarray\ iarray \Rightarrow nat$
where $ncols\text{-}iarray\ A = IArray.length\ (A!!0)$

definition $rows\text{-}iarray\ A = \{row\text{-}iarray\ i\ A \mid i.\ i \in \{..\lt nrows\text{-}iarray\ A\}\}$

definition $columns\text{-}iarray\ A = \{column\text{-}iarray\ i\ A \mid i.\ i \in \{..\lt ncols\text{-}iarray\ A\}\}$

definition $tabulate2 :: nat \Rightarrow nat \Rightarrow (nat \Rightarrow nat \Rightarrow 'a) \Rightarrow 'a\ iarray\ iarray$
where $tabulate2\ m\ n\ f = IArray.of\text{-}fun\ (\lambda i.\ IArray.of\text{-}fun\ (f\ i)\ n)\ m$

definition $transpose\text{-}iarray :: 'a\ iarray\ iarray \Rightarrow 'a\ iarray\ iarray$
where $transpose\text{-}iarray\ A = tabulate2\ (ncols\text{-}iarray\ A)\ (nrows\text{-}iarray\ A)\ (\lambda a\ b.\ A!!b!!a)$

definition $matrix\text{-}matrix\text{-}mult\text{-}iarray :: 'a::\{times,\ comm\text{-}monoid\text{-}add\}\ iarray\ iarray \Rightarrow 'a\ iarray\ iarray \Rightarrow 'a\ iarray\ iarray$ (**infixl** $**i\ 70$)
where $A\ **i\ B = tabulate2\ (nrows\text{-}iarray\ A)\ (ncols\text{-}iarray\ B)\ (\lambda i\ j.\ sum\ (\lambda k.\ ((A!!i)!!k) * ((B!!k)!!j))\ \{0..\lt ncols\text{-}iarray\ A\})$

definition $matrix\text{-}vector\text{-}mult\text{-}iarray :: 'a::\{semiring\text{-}1\}\ iarray\ iarray \Rightarrow 'a\ iarray \Rightarrow 'a\ iarray$ (**infixl** $*iv\ 70$)
where $A\ *iv\ x = IArray.of\text{-}fun\ (\lambda i.\ sum\ (\lambda j.\ ((A!!i)!!j) * (x!!j))\ \{0..\lt IArray.length\ x\})\ (nrows\text{-}iarray\ A)$

definition $vector\text{-}matrix\text{-}mult\text{-}iarray :: 'a::\{semiring\text{-}1\}\ iarray \Rightarrow 'a\ iarray\ iarray \Rightarrow 'a\ iarray$ (**infixl** $v*i\ 70$)
where $x\ v*i\ A = IArray.of\text{-}fun\ (\lambda j.\ sum\ (\lambda i.\ ((A!!i)!!j) * (x!!i))\ \{0..\lt IArray.length\ x\})\ (ncols\text{-}iarray\ A)$

definition $mat\text{-}iarray :: 'a::\{zero\} \Rightarrow nat \Rightarrow 'a\ iarray\ iarray$
where $mat\text{-}iarray\ k\ n = tabulate2\ n\ n\ (\lambda i\ j.\ if\ i = j\ then\ k\ else\ 0)$

definition $is\text{-}zero\text{-}iarray :: 'a::\{zero\}\ iarray \Rightarrow bool$
where $is\text{-}zero\text{-}iarray\ A = IArray\text{-}Addenda\text{-}QR.all\ (\lambda i.\ A!!i = 0)\ (IArray[0..\lt IArray.length\ A])$

9.2.1 Properties of previous definitions

lemma $is\text{-}zero\text{-}iarray\text{-}eq\text{-}iff$:

fixes $A::'a::\{zero\} \wedge n::\{mod\text{-}type\}$

shows $(A = 0) = (is\text{-}zero\text{-}iarray\ (vec\text{-}to\text{-}iarray\ A))$

proof (*auto*)

show $is\text{-}zero\text{-}iarray\ (vec\text{-}to\text{-}iarray\ 0)$ **by** (*simp add: vec-to-iarray-def is-zero-iarray-def Option.is-none-def find-None-iff*)

show $is\text{-}zero\text{-}iarray\ (vec\text{-}to\text{-}iarray\ A) \Longrightarrow A = 0$

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 $\forall i \in \{0..<CARD('n)\}$. $A \ \$ \ mod\text{-}type\text{-}class.from\text{-}nat \ i = 0$
hence $eq\text{-}zero: \forall x < CARD('n). A \ \$ \ from\text{-}nat \ x = 0$ **by** *force*
have $to\text{-}nat \ i < CARD('n)$ **using** *bij-to-nat*[**where** $?'a='n$] **unfolding** *bij-betw-def*
by *fastforce*
hence $A \ \$ \ (from\text{-}nat \ (to\text{-}nat \ i)) = 0$ **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** $mult\text{-}iarray \ A \ q \ !! \ a = q * A!!a$
unfolding *mult-iarray-def*
unfolding *IArray.of-fun-def* **unfolding** *sub-def*
using *assms* **by** *simp*

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

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

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

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

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

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

lemma *vec-to-iarray-row*[*code-unfold*]: $vec\text{-}to\text{-}iarray \ (row \ i \ A) = row\text{-}iarray \ (to\text{-}nat \ i) \ (matrix\text{-}to\text{-}iarray \ A)$
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'*: $\text{vec-to-iarray} (\text{row } i \ A) = (\text{matrix-to-iarray } A) !! (\text{to-nat } i)$

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*]: $\text{vec-to-iarray} (\text{column } i \ A) = \text{column-iarray} (\text{to-nat } i) (\text{matrix-to-iarray } A)$

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 $(\text{vec-to-iarray} (\text{column} (\text{from-nat } k) \ A)) = (\text{column-iarray } k (\text{matrix-to-iarray } A))$

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 $\text{column-iarray } j \ A !! i = A !! i !! j$

proof –

have $\text{column-iarray } j \ A !! i = \text{map} (\lambda m. A !! m !! j) [0..<IArray.length \ A] ! i$

unfolding *column-iarray-def* **by** *auto*

also have $\dots = (\lambda m. A !! m !! j) ([0..<IArray.length \ A] ! i)$ **using** *i nth-map*

unfolding *nrows-iarray-def* **by** *auto*

also have $\dots = (\lambda m. A !! m !! j) (i)$ **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*: $\text{vec-to-iarray}' (\text{rows } A) = \text{rows-iarray} (\text{matrix-to-iarray } A)$

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*: $\text{vec-to-iarray}' (\text{columns } A) = \text{columns-iarray} (\text{matrix-to-iarray } A)$

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*)

9.3 Definition of elementary operations

definition *interchange-rows-iarray* :: $'a \ \text{iarray} \ \text{iarray} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow 'a \ \text{iarray} \ \text{iarray}$

where *interchange-rows-iarray* $A \ a \ b = IArray.of-fun (\lambda n. \text{if } n=a \ \text{then } A!!b \ \text{else if } n=b \ \text{then } A!!a \ \text{else } A!!n) (IArray.length \ A)$

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, zero} 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)

9.3.1 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* *Let-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 Let-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-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 $A::'a::\{\text{semiring-1}\}^{\wedge m}::\{\text{mod-type}\}^{\wedge n}::\{\text{mod-type}\}$ and $B::'a^{\wedge b}::\{\text{mod-type}\}^{\wedge m}::\{\text{mod-type}\}$
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-cong*[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 $A::'a::\{\text{semiring-1}\}^{\wedge m}::\{\text{mod-type}\}^{\wedge n}::\{\text{mod-type}\}$ and $x::'a^{\wedge m}::\{\text{mod-type}\}$
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-cong*[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 $A::'a::\{\text{semiring-1}\}^{\wedge m}::\{\text{mod-type}\}^{\wedge n}::\{\text{mod-type}\}$ and $x::'a^{\wedge n}::\{\text{mod-type}\}$
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
show $(\sum i \in \text{UNIV}. A \$ i \$ \text{from-nat } xa * x \$ i) = (\sum i = 0..< \text{CARD}('n). A \$ \text{from-nat } i \$ \text{from-nat } xa * x \$ \text{from-nat } i)$
apply (*rule sum.reindex-cong*[of *from-nat::nat=>'n*]) **using** *bij-from-nat*[**where** *?'a='n*] **unfolding** *bij-betw-def* **by** *fast+*
qed

lemma *matrix-to-iarray-interchange-rows*[code-unfold]:
fixes $A::'a::\{\text{semiring-1}\}^{\wedge \text{columns}}::\{\text{mod-type}\}^{\wedge \text{rows}}::\{\text{mod-type}\}$
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*: $x < \text{CARD}('rows)$
and *x-not-j*: $x \neq \text{to-nat } j$ and *x-not-i*: $x \neq \text{to-nat } i$
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 $A::'a::\{\text{semiring-1}\}^{\sim}\text{columns}::\{\text{mod-type}\}^{\sim}\text{rows}::\{\text{mod-type}\}$
 shows $\text{matrix-to-iarray} (\text{mult-row } A \ i \ q) = \text{mult-row-iarray} (\text{matrix-to-iarray } A)$
 (*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*: $i \neq \text{to-nat} (\text{from-nat } i::\text{rows})$ **and** $x < \text{CARD}(\text{columns})$
 and $i < \text{CARD}(\text{rows})$
 hence $i = \text{to-nat} (\text{from-nat } i::\text{rows})$ **using** *to-nat-from-nat-id* **by** *fastforce*
 thus $q * A \ \$ \ \text{from-nat } i \ \$ \ \text{from-nat } x = A \ \$ \ \text{from-nat } i \ \$ \ \text{from-nat } x$
 using *i-contr* **by** *contradiction*
qed

lemma *matrix-to-iarray-row-add*[code-unfold]:
 fixes $A::'a::\{\text{semiring-1}\}^{\sim}\text{columns}::\{\text{mod-type}\}^{\sim}\text{rows}::\{\text{mod-type}\}$
 shows $\text{matrix-to-iarray} (\text{row-add } A \ i \ j \ q) = \text{row-add-iarray} (\text{matrix-to-iarray } A)$
 (*to-nat i*) (*to-nat j*) *q*
proof (*unfold matrix-to-iarray-def row-add-iarray-def o-def, auto*)
 show $\text{vec-to-iarray} (\text{row-add } A \ i \ j \ q \ \$ \ i) = \text{vec-to-iarray} (A \ \$ \ i) + \text{mult-iarray}$
 (*vec-to-iarray* ($A \ \$ \ j$)) *q*
 unfolding *mult-iarray-def vec-to-iarray-def* **unfolding** *plus-iarray-def Let-def*
 row-add-def **by** *auto*
 fix *ia* **assume** *ia-not-i*: $ia \neq \text{to-nat } i$ **and** *ia-card*: $ia < \text{CARD}(\text{rows})$
 have *from-nat-ia-not-i*: $\text{from-nat } ia \neq i$
 proof (*rule ccontr*)
 assume $\neg \text{from-nat } ia \neq i$ **hence** $\text{from-nat } ia = i$ **by** *simp*
 hence $\text{to-nat} (\text{from-nat } ia::\text{rows}) = \text{to-nat } i$ **by** *simp*
 hence $ia = \text{to-nat } i$ **using** *to-nat-from-nat-id ia-card* **by** *fastforce*
 thus *False* **using** *ia-not-i* **by** *contradiction*
 qed
 show $\text{vec-to-iarray} (\text{row-add } A \ i \ j \ q \ \$ \ \text{from-nat } ia) = \text{vec-to-iarray} (A \ \$ \ \text{from-nat}$
 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 $A::'a::\{\text{semiring-1}\}^{\sim}\text{columns}::\{\text{mod-type}\}^{\sim}\text{rows}::\{\text{mod-type}\}$
 shows $\text{matrix-to-iarray} (\text{interchange-columns } A \ i \ j) = \text{interchange-columns-iarray}$
 (*matrix-to-iarray* A) (*to-nat i*) (*to-nat j*)
 unfolding *interchange-columns-def interchange-columns-iarray-def o-def tabu-*
 late2-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

10 Gram Schmidt over IArrays

theory *Gram-Schmidt-IArrays*
imports
QR-Decomposition
Matrix-To-IArray-QR
begin

10.1 Some previous definitions, lemmas and instantiations about 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])*

instantiation *iarray* :: (*scaleR*) *scaleR*
begin
definition *scaleR-iarray* *k A = IArray.of-fun (λi. k *_R (A !! i)) (IArray.length A)*
instance proof qed
end

instantiation *iarray* :: (*times*) *times*
begin

definition *times-iarray* $A B = IArray.of\text{-}fun (\lambda i. A!!i * B !! i) (IArray.length A)$
instance proof qed
end

lemma *plus-iarray-component*:

assumes $iA: i < IArray.length A$
and $iB: i < IArray.length B$
shows $(A+B) !! i = A!!i + B!!i$
proof (*unfold plus-iarray-def Let-def*)
have $IArray.of\text{-}fun$
 $(\lambda a. IArray.of\text{-}fun (\lambda a. \text{if } a < IArray.length A \text{ then } A !! a \text{ else } 0) (\max$
 $(IArray.length A) (IArray.length B)) !! a +$
 $IArray.of\text{-}fun (\lambda a. \text{if } a < IArray.length B \text{ then } B !! a \text{ else } 0) (\max (IArray.length$
 $A) (IArray.length B)) !! a)$
 $(\max (IArray.length A) (IArray.length B)) !!$
 $i = IArray.of\text{-}fun (\lambda a. \text{if } a < IArray.length A \text{ then } A !! a \text{ else } 0) (\max$
 $(IArray.length A) (IArray.length B)) !! i +$
 $IArray.of\text{-}fun (\lambda a. \text{if } a < IArray.length B \text{ then } B !! a \text{ else } 0) (\max (IArray.length$
 $A) (IArray.length B)) !! i$
by (*rule of-fun-nth, metis iB less-max-iff-disj*)
also have $... = (\lambda a. \text{if } a < IArray.length A \text{ then } A !! a \text{ else } 0) i +$
 $(\lambda a. \text{if } a < IArray.length B \text{ then } B !! a \text{ else } 0) i$
using *of-fun-nth*[*of* $i (\max (IArray.length A) (IArray.length B))$] **using** iB **by**
simp
also have $... = A!!i + B !! i$ **using** iA iB **by** *simp*
finally show $IArray.of\text{-}fun$
 $(\lambda a. IArray.of\text{-}fun (\lambda a. \text{if } a < IArray.length A \text{ then } A !! a \text{ else } 0) (\max$
 $(IArray.length A) (IArray.length B)) !! a +$
 $IArray.of\text{-}fun (\lambda a. \text{if } a < IArray.length B \text{ then } B !! a \text{ else } 0) (\max (IArray.length$
 $A) (IArray.length B)) !! a)$
 $(\max (IArray.length A) (IArray.length B)) !!$
 $i =$
 $A !! i + B !! i .$

qed

lemma *minus-iarray-component*:

assumes $iA: i < IArray.length A$
and $iB: i < IArray.length B$
shows $(A-B) !! i = A!!i - B!!i$
proof (*unfold minus-iarray-def Let-def*)
have $IArray.of\text{-}fun$
 $(\lambda a. IArray.of\text{-}fun (\lambda a. \text{if } a < IArray.length A \text{ then } A !! a \text{ else } 0) (\max$
 $(IArray.length A) (IArray.length B)) !! a -$
 $IArray.of\text{-}fun (\lambda a. \text{if } a < IArray.length B \text{ then } B !! a \text{ else } 0) (\max (IArray.length$
 $A) (IArray.length B)) !! a)$
 $(\max (IArray.length A) (IArray.length B)) !!$
 $i = IArray.of\text{-}fun (\lambda a. \text{if } a < IArray.length A \text{ then } A !! a \text{ else } 0) (\max$
 $(IArray.length A) (IArray.length B)) !! i -$

```

  IArray.of-fun (λa. if a < IArray.length B then B !! a else 0) (max (IArray.length
A) (IArray.length B)) !! i
  by (rule of-fun-nth, metis iB less-max-iff-disj)
  also have ... = (λa. if a < IArray.length A then A !! a else 0) i -
    (λa. if a < IArray.length B then B !! a else 0) i
  using of-fun-nth[of i (max (IArray.length A) (IArray.length B))] using iB by
simp
  also have ... = A !! i - B !! i using iA iB by simp
  finally show IArray.of-fun
    (λa. IArray.of-fun (λa. if a < IArray.length A then A !! a else 0) (max
(IArray.length A) (IArray.length B)) !! a -
    IArray.of-fun (λa. if a < IArray.length B then B !! a else 0) (max (IArray.length
A) (IArray.length B)) !! a)
    (max (IArray.length A) (IArray.length B)) !! i = A !! i - B !! i .
qed

```

lemma *length-plus-iarray*:

```

  IArray.length (A+B) = max (IArray.length A) (IArray.length B)
  unfolding plus-iarray-def Let-def by auto

```

lemma *length-sum-iarray*:

```

  assumes finite S and S ≠ {}
  shows IArray.length (sum f S) = Max {IArray.length (f x) | x. x ∈ S}
  using assms
proof (induct S, simp)
  case (insert x F)
  show ?case
proof (cases F = {})
  case True show ?thesis unfolding True by auto
next
  case False
  have rw: IArray.length (sum f F) = Max {IArray.length (f x) | x. x ∈ F}
  by (rule insert.hyps, simp add: False)
  have set-rw: (insert (IArray.length (f x)) {IArray.length (f x) | x. x ∈ F}) =
{IArray.length (f a) | a. a ∈ insert x F}
  by auto
  have IArray.length (sum f (insert x F)) = IArray.length (f x + sum f F)
  by (metis insert.hyps(1) insert.hyps(2) sum-clauses(2))
  also have ... = max (IArray.length (f x)) (IArray.length (sum f F))
  unfolding length-plus-iarray ..
  also have ... = max (IArray.length (f x)) (Max {IArray.length (f a) | a. a ∈ F})
  unfolding rw by simp
  also have ... = Max (insert (IArray.length (f x)) {IArray.length (f a) | a. a ∈
F})
  proof(rule Max-insert[symmetric])
  show finite {IArray.length (f x) | x. x ∈ F} using insert.hyps(1) by auto
  show {IArray.length (f x) | x. x ∈ F} ≠ {} by (metis (lifting, mono-tags) False
empty-Collect-eq ex-in-conv)

```

```

qed
also have ...= Max {IArray.length (f a) | a. a ∈ insert x F} unfolding set-rw
..
finally show ?thesis .
qed
qed

```

lemma *sum-component-iarray*:

```

assumes a: ∀ x∈S. i < IArray.length (f x)
and f: finite S
and S: S ≠ {} — If S is empty, then the sum will return the empty iarray and it
makes no sense to access the component i
shows sum f S !! i = (∑ x∈S. f x !! i)
using f a S
proof (induct S, simp)
case (insert x F)
have finite-F: finite F by (metis insert.hyps(1))
show ?case
proof (cases F={})
case True
have sum f (insert x F) !! i = f x !! i unfolding True by auto
also have ... = (∑ x∈insert x F. f x !! i) unfolding True by auto
finally show ?thesis .
next
case False
have hyp: (sum f F !! i) = (∑ x∈F. f x !! i)
proof (rule insert.hyps)
show ∀ x∈F. i < IArray.length (f x) by (metis insert.prem(1) insertCI)
show F ≠ {} using False .
qed
have sum f (insert x F) !! i = (f x + sum f F) !! i
by (metis insert.hyps(1) insert.hyps(2) sum-clauses(2))
also have ... = (f x) !! i + (sum f F !! i)
proof (rule plus-iarray-component)
obtain a where a: a ∈ F using False by auto
have finite-C: finite {IArray.length (f x) | x. x ∈ F} using finite-F by auto
have not-empty-C: {IArray.length (f x) | x. x ∈ F} ≠ {} using False by simp

show i < IArray.length (f x) by (metis insert.prem(1) insertI1)
show i < IArray.length (sum f F)
unfolding length-sum-iarray[OF finite-F False]
unfolding Max-gr-iff[OF finite-C not-empty-C]
proof (rule bexI[of - IArray.length (f a)])
show i < IArray.length (f a) using insert.prem(1) a by auto
show IArray.length (f a) ∈ {IArray.length (f x) | x. x ∈ F} using a by auto
qed
qed
also have ... = (f x) !! i + (∑ x∈F. f x !! i) unfolding hyp ..

```

also have ... = $(\sum x \in \text{insert } x \text{ } F. f x !! i)$
by (*metis* (*mono-tags*) *insert.hyps(1)* *insert.hyps(2)* *sum-clauses(2)*)
finally show $\text{sum } f \text{ (insert } x \text{ } F) !! i = (\sum x \in \text{insert } x \text{ } F. f x !! i)$.
qed
qed

lemma *length-zero-iarray*: $IArray.length \ 0 = 0$
unfolding *zero-iarray-def* **by** *simp*

lemma *minus-zero-iarray*:
fixes $A :: 'a :: \{group\text{-add}\}$ *iarray*
shows $A - 0 = A$
proof (*unfold* *iarray-exhaust2* *list-eq-iff-nth-eq*, *auto*, *unfold* *IArray.length-def[symmetric]* *IArray.sub-def[symmetric]*)
have *max-eq*: $(\max (IArray.length \ A) (IArray.length \ 0)) = IArray.length \ A$
unfolding *zero-iarray-def* **by** *auto*
show *length-eq*: $IArray.length \ (A - 0) = IArray.length \ A$
unfolding *minus-iarray-def* *Let-def* **unfolding** *max-eq* **by** *auto*
fix *i* **assume** $i < IArray.length \ (A - 0)$
hence *i2*: $i < (IArray.length \ A)$ **unfolding** *length-eq* .
have $A - 0 = IArray.of\text{-fun}$
 $(\lambda a. IArray.of\text{-fun} \ (\lambda a. \text{if } a < IArray.length \ A \text{ then } A !! a \text{ else } 0) \ (IArray.length \ A) !! a -$
 $IArray.of\text{-fun} \ (\lambda a. \text{if } a < IArray.length \ (0 :: 'a \ iarray) \text{ then } 0 !! a \text{ else } 0) \ (IArray.length \ A) !! a)$
 $(IArray.length \ A)$
unfolding *minus-iarray-def* *Let-def* **unfolding** *max-eq* ..
also have ... !! $i = A !! i$ **unfolding** *of-fun-nth[OF i2]* *length-zero-iarray* **using** *i2* **by** *auto*
finally show $(A - 0) !! i = A !! i$.
qed

10.2 Inner mult over real iarrays

definition *inner-iarray* :: *real iarray* => *real iarray* => *real* (**infixl** \cdot *i* 70)
where $\text{inner-iarray } A \ B = \text{sum} \ (\lambda n. A !! n * B !! n) \ \{0..<IArray.length \ A\}$

lemma *vec-to-iarray-inner*:
 $a \cdot b = \text{vec-to-iarray } a \cdot i \ \text{vec-to-iarray } b$
proof (*unfold* *inner-iarray-def* *inner-vec-def*, *auto*, *unfold* *IArray.sub-def[symmetric]* *IArray.length-def[symmetric]*)
have *set-rw*: $\{0..<IArray.length \ (\text{vec-to-iarray } a)\} = (\text{to-nat}) \ (\text{UNIV} :: 'a \ \text{set})$
unfolding *vec-to-iarray-def*
using *to-nat-less-card* **using** *bij-to-nat[where ?'a='a]*
unfolding *bij-betw-def* **by** *auto*
have *inj*: $\text{inj-on} \ (\text{to-nat} :: 'a \Rightarrow \text{nat}) \ (\text{UNIV} :: 'a \ \text{set})$
by (*metis* *strict-mono-imp-inj-on* *strict-mono-to-nat*)
have $(\sum n = 0..<IArray.length \ (\text{vec-to-iarray } a). \ \text{vec-to-iarray } a !! n * \ \text{vec-to-iarray } b !! n)$

= $(\sum n \in \text{range } (to\text{-nat}::'a \Rightarrow nat). \text{vec-to-iarray } a !! n * \text{vec-to-iarray } b !! n)$
unfolding *set-rw* ..
also have ... = $(\sum x \in (UNIV::'a \text{ set}). \text{vec-to-iarray } a !! to\text{-nat } x * \text{vec-to-iarray } b !! to\text{-nat } x)$
unfolding *sum.reindex[OF inj] o-def* ..
also have ... = $(\sum x \in UNIV. a \$ x * b \$ x)$ **unfolding** *vec-to-iarray-nth'* ..
finally show $(\sum i \in UNIV. a \$ i * b \$ i)$
 = $(\sum n = 0..<IArray.length (\text{vec-to-iarray } a). \text{vec-to-iarray } a !! n * \text{vec-to-iarray } b !! n)$..
qed

lemma *vec-to-iarray-scaleR*:

vec-to-iarray $(a *_R x) = a *_R (\text{vec-to-iarray } x)$

unfolding *scaleR-vec-def scaleR-iarray-def vec-to-iarray-def* **by** *auto*

10.3 Gram Schmidt over IArrays

definition *Gram-Schmidt-column-k-iarrays* $A \ k$

= *tabulate2* $(nrows\text{-iarray } A) (ncols\text{-iarray } A) (\lambda a \ b. (if \ b = k$
 then $(column\text{-iarray } b \ A - sum (\lambda x. (((column\text{-iarray } b \ A) \cdot i \ x) / (x \cdot i \ x)) *_R$
 $x)$
 (set $(List.map (\lambda n. column\text{-iarray } n \ A) [0..<b])))$
 else $(column\text{-iarray } b \ A) !! a)$

definition *Gram-Schmidt-upt-k-iarrays* $A \ k = List.foldl \text{Gram-Schmidt-column-k-iarrays} \ A \ [0..<(Suc \ k)]$

definition *Gram-Schmidt-matrix-iarrays* $A = \text{Gram-Schmidt-upt-k-iarrays } A \ (ncols\text{-iarray } A - 1)$

lemma *matrix-to-iarray-Gram-Schmidt-column-k*:

fixes $A::real \wedge cols::\{mod\text{-type}\} \wedge rows::\{mod\text{-type}\}$

assumes $k: k < ncols \ A$

shows $matrix\text{-to-iarray} (\text{Gram-Schmidt-column-k } A \ k) = \text{Gram-Schmidt-column-k-iarrays} (matrix\text{-to-iarray } A) \ k$

proof (*unfold iarray-exhaust2 list-eq-iff-nth-eq, rule conjI, auto, unfold IArray.sub-def[symmetric] IArray.length-def[symmetric]*)

show $IArray.length (matrix\text{-to-iarray} (\text{Gram-Schmidt-column-k } A \ k)) = IArray.length (\text{Gram-Schmidt-column-k-iarrays} (matrix\text{-to-iarray } A) \ k)$

unfolding *matrix-to-iarray-def Gram-Schmidt-column-k-iarrays-def tabulate2-def*

unfolding *nrows-iarray-def* **by** *auto*

fix $i: i < IArray.length (matrix\text{-to-iarray} (\text{Gram-Schmidt-column-k } A \ k))$

show $IArray.length (matrix\text{-to-iarray} (\text{Gram-Schmidt-column-k } A \ k) !! i)$

= $IArray.length (\text{Gram-Schmidt-column-k-iarrays} (matrix\text{-to-iarray } A) \ k !! i)$

unfolding *matrix-to-iarray-def Gram-Schmidt-column-k-iarrays-def tabulate2-def*

unfolding *nrows-iarray-def ncols-iarray-def o-def*

proof (*auto*)

```

have f1:  $i < \text{card } (\text{UNIV}::\text{'rows set})$ 
by (metis  $i$  length-eq-card-rows)
have f2:  $\bigwedge x_5. \text{IArray.list-of } (\text{vec-to-iarray } x_5)$ 
  =  $\text{List.map } (\lambda uua. x_5 \$ (\text{from-nat } uua::\text{'cols})::\text{real}) [0..<\text{card } (\text{UNIV}::\text{'cols set})]$ 
by (metis list-of.simps IArray.of-fun-def vec-to-iarray-def)
thus length (IArray.list-of (List.map ( $\lambda x. \text{vec-to-iarray } (\text{Gram-Schmidt-column-k } A \ k \ \$ \ \text{from-nat } x)$ ) [0..<card (UNIV::'rows set)] !  $i$ ))
  = length (IArray.list-of (List.map ( $\lambda i. \text{IArray } (\text{List.map } (\lambda b. \text{IArray.list-of } (\text{if } b = k \ \text{then } \text{column-iarray } b \ (\text{IArray } (\text{List.map } (\lambda x. \text{vec-to-iarray } (A \ \$ \ \text{from-nat } x)) [0..<\text{card } (\text{UNIV}::\text{'rows set})))$ 
    - ( $\sum x \in \text{set } (\text{List.map } (\lambda n. \text{column-iarray } n \ (\text{IArray } (\text{List.map } (\lambda x. \text{vec-to-iarray } (A \ \$ \ \text{from-nat } x)) [0..<\text{card } (\text{UNIV}::\text{'rows set})))) [0..< b]). (\text{column-iarray } b \ (\text{IArray } (\text{List.map } (\lambda x. \text{vec-to-iarray } (A \ \$ \ \text{from-nat } x)) [0..<\text{card } (\text{UNIV}::\text{'rows set}))) \cdot i \ x / (x \cdot i \ x) *_{\mathbb{R}} x)$  else  $\text{column-iarray } b \ (\text{IArray } (\text{List.map } (\lambda x. \text{vec-to-iarray } (A \ \$ \ \text{from-nat } x)) [0..<\text{card } (\text{UNIV}::\text{'rows set})))) ! i$ ) [0..<length (IArray.list-of (vec-to-iarray (A $ from-nat 0))))]) [0..<card (UNIV::'rows set)] !  $i$ ))
    using f1 by auto
qed
next
fix  $i \ ia$ 
assume  $i: i < \text{IArray.length } (\text{matrix-to-iarray } (\text{Gram-Schmidt-column-k } A \ k))$ 
and  $ia: ia < \text{IArray.length } (\text{matrix-to-iarray } (\text{Gram-Schmidt-column-k } A \ k))$  !!
 $i$ )
have  $i\text{-nrows}: i < \text{nrows } A$  using  $i$  unfolding matrix-to-iarray-def nrows-def by
auto
have  $ia\text{-ncols}: ia < \text{ncols } A$  using  $ia$  unfolding matrix-to-iarray-def o-def vec-to-iarray-def
ncols-def
by (auto, metis (no-types) Ex-list-of-length  $i\text{-nrows}$  length-map list-of.simps
map-nth nrows-def nth-map)
have  $i\text{-nrows-iarray}: i < \text{nrows-iarray } (\text{matrix-to-iarray } A)$  using  $i\text{-nrows}$  by
(metis matrix-to-iarray-nrows)
have  $ia\text{-ncols-iarray}: ia < \text{ncols-iarray } (\text{matrix-to-iarray } A)$  using  $ia\text{-ncols}$  by
(metis matrix-to-iarray-ncols)
show matrix-to-iarray (Gram-Schmidt-column-k A k) !!  $i$  !!  $ia = \text{Gram-Schmidt-column-k-iarrays}$ 
(matrix-to-iarray A) k !!  $i$  !!  $ia$ 
unfolding Gram-Schmidt-column-k-def Gram-Schmidt-column-k-iarrays-def
unfolding matrix-to-iarray-nth[of - from-nat  $i::\text{'rows from-nat } ia::\text{'cols}$ ,
unfolded to-nat-from-nat-id[OF  $i\text{-nrows}$ [unfolded nrows-def]]
to-nat-from-nat-id[OF  $ia\text{-ncols}$ [unfolded ncols-def]]]
unfolding tabulate2-def
unfolding of-fun-nth[OF  $i\text{-nrows-iarray}$ ]
unfolding of-fun-nth[OF  $ia\text{-ncols-iarray}$ ]
proof (unfold proj-onto-def proj-def[abs-def], auto, unfold IArray.sub-def[symmetric])
have  $\text{inj}: \text{inj-on } \text{vec-to-iarray } \{\text{column } i \ A \mid i. i < \text{from-nat } k\}$  unfolding
inj-on-def
by (auto, metis vec-to-iarray-morph)
have  $\text{set-rw}: \{\text{column } i \ A \mid i. i < \text{from-nat } k\} = (\lambda n. \text{column } n \ A) ' \{0..<\text{from-nat } k\}$ 

```

```

proof (unfold image-def, auto)
  fix a::'cols assume a: a < from-nat k
  show  $\exists x \in \{0..<from\text{-}nat\ k\}. column\ a\ A = column\ x\ A$ 
    by (rule bezI[of - a], auto simp add: a least-mod-type)
qed
have set-rw2: vec-to-iarray ' {column i A | i. i < from-nat k}
  = ( $\lambda n. column\text{-}iarray\ n\ (matrix\text{-}to\text{-}iarray\ A)$ ) ' {0..<k}
proof (unfold image-def, auto)
  fix a::'cols assume a: a < from-nat k
  show  $\exists x \in \{0..<k\}. vec\text{-}to\text{-}iarray\ (column\ a\ A) = column\text{-}iarray\ x\ (matrix\text{-}to\text{-}iarray\ A)$ 
    by (rule bezI[of - to-nat a], auto simp add: a to-nat-le vec-to-iarray-column)
next
  fix xa assume xa: xa < k
  have xa': (from-nat xa::'cols) < from-nat k by (rule from-nat-mono[OF xa
k[unfolded ncols-def]])
  show  $\exists x. (\exists i. x = column\ i\ A \wedge i < from\text{-}nat\ k) \wedge column\text{-}iarray\ xa\ (matrix\text{-}to\text{-}iarray\ A) = vec\text{-}to\text{-}iarray\ x$ 
    apply (rule exI[of - column (from-nat xa) A])
    apply auto apply (rule exI[of - from-nat xa])
    apply (auto simp add: xa' vec-to-iarray-column)
    by (metis k order.strict-trans vec-to-iarray-column vec-to-iarray-column' xa)
qed
show column (from-nat k) A $ from-nat i -
  ( $\sum x \in \{column\ i\ A \mid i. i < from\text{-}nat\ k\}. column\ (from\text{-}nat\ k)\ A \cdot x * x\ \$\ from\text{-}nat\ i / (x \cdot x)$ ) =
  (column-iarray k (matrix-to-iarray A) -
  ( $\sum x \in (\lambda n. column\text{-}iarray\ n\ (matrix\text{-}to\text{-}iarray\ A))\ ' \{0..<k\}. (column\text{-}iarray\ k\ (matrix\text{-}to\text{-}iarray\ A) \cdot i\ x / (x \cdot i\ x)) *_R\ x$ ) !! i)
proof (cases k=0)
  case True
  have set-rw-empty: {column i A | i. i < from-nat k}={ }
    unfolding True from-nat-0 using least-mod-type not-le by auto
  have column (from-nat k) A $ from-nat i -
  ( $\sum x \in \{column\ i\ A \mid i. i < from\text{-}nat\ k\}. column\ (from\text{-}nat\ k)\ A \cdot x * x\ \$\ from\text{-}nat\ i / (x \cdot x)$ ) =
  column (from-nat k) A $ from-nat i - 0 unfolding set-rw-empty by simp
  also have ...= column (from-nat k) A $ from-nat i by simp
  also have ...= (column-iarray k (matrix-to-iarray A) - 0) !! i
    unfolding minus-zero-iarray
    unfolding vec-to-iarray-column'[OF k, symmetric]
    unfolding vec-to-iarray-nth[OF i-nrows[unfolded nrows-def]] ..
  also have ...= (column-iarray k (matrix-to-iarray A) -
  ( $\sum x \in (\lambda n. column\text{-}iarray\ n\ (matrix\text{-}to\text{-}iarray\ A))\ ' \{0..<k\}. (column\text{-}iarray\ k\ (matrix\text{-}to\text{-}iarray\ A) \cdot i\ x / (x \cdot i\ x)) *_R\ x$ )
  !! i) unfolding True by auto
  finally show ?thesis .
next
  case False

```


have $(\sum x \in (\lambda n. \text{column-iarray } n \text{ (matrix-to-iarray } A)) \text{ ' } \{0..<k\}$. $(\text{column-iarray } k \text{ (matrix-to-iarray } A) \cdot i \text{ } x / (x \cdot i \text{ } x)) *_R x \text{ !! } i =$
 $(\sum x \in (\lambda n. \text{column-iarray } n \text{ (matrix-to-iarray } A)) \text{ ' } \{0..<k\}$.
 $((\text{column-iarray } k \text{ (matrix-to-iarray } A) \cdot i \text{ } x / (x \cdot i \text{ } x)) *_R x \text{ !! } i)$
proof $(\text{rule sum-component-iarray})$
show $\forall x \in (\lambda n. \text{column-iarray } n \text{ (matrix-to-iarray } A)) \text{ ' } \{0..<k\}$. $i < \text{IArray.length } ((\text{column-iarray } k \text{ (matrix-to-iarray } A) \cdot i \text{ } x / (x \cdot i \text{ } x)) *_R x)$
proof $(\text{unfold column-iarray-def, auto})$
fix $x :: \text{nat}$
have $i < \text{length (IArray.list-of (IArray (map (vec-to-iarray } \circ (\$) A \circ$
 $\text{mod-type-class.from-nat) [0..<card (UNIV::'rows set}])))$
by $(\text{metis i-nrows-iarray IArray.length-def matrix-to-iarray-def nrows-iarray-def})$
thus $i < \text{length (IArray.list-of ((IArray (map (\lambda n. \text{IArray.list-of (IArray.list-of$
 $(\text{matrix-to-iarray } A) ! n) ! k) [0..<\text{length (IArray.list-of (matrix-to-iarray } A))])$
 $\cdot i \text{ IArray (map (\lambda n. \text{IArray.list-of (IArray.list-of (matrix-to-iarray } A) ! n) ! x)$
 $[0..<\text{length (IArray.list-of (matrix-to-iarray } A))]) / (IArray (map (\lambda n. \text{IArray.list-of$
 $(\text{IArray.list-of (matrix-to-iarray } A) ! n) ! x) [0..<\text{length (IArray.list-of (matrix-to-iarray$
 $A))]) \cdot i \text{ IArray (map (\lambda n. \text{IArray.list-of (IArray.list-of (matrix-to-iarray } A) ! n) !$
 $x) [0..<\text{length (IArray.list-of (matrix-to-iarray } A))])]) *_R \text{IArray (map (\lambda n. \text{IArray.list-of$
 $(\text{IArray.list-of (matrix-to-iarray } A) ! n) ! x) [0..<\text{length (IArray.list-of$
 $(\text{matrix-to-iarray } A))])])$
by $(\text{simp add: matrix-to-iarray-def scaleR-iarray-def})$
qed
show $\text{finite } ((\lambda n. \text{column-iarray } n \text{ (matrix-to-iarray } A)) \text{ ' } \{0..<k\})$ **by** auto
show $(\lambda n. \text{column-iarray } n \text{ (matrix-to-iarray } A)) \text{ ' } \{0..<k\} \neq \{\}$ **using**
 False by auto
qed
also have $\dots = (\sum x \in (\lambda n. \text{column-iarray } n \text{ (matrix-to-iarray } A)) \text{ ' } \{0..<k\}$.
 $(\text{column-iarray } k \text{ (matrix-to-iarray } A) \cdot i \text{ } x / (x \cdot i \text{ } x)) *_R x \text{ !! } i)$
proof (rule sum.cong)
fix x **assume** $x \in (\lambda n. \text{column-iarray } n \text{ (matrix-to-iarray } A)) \text{ ' } \{0..<k\}$
from this obtain n **where** $x = \text{column-iarray } n \text{ (matrix-to-iarray } A)$
and $n: n < k$ **by** auto
have $n\text{-ncols}: n < \text{ncols } A$ **by** $(\text{metis } k \text{ } n \text{ order.strict-trans})$
have $c\text{-eq}: \text{column-iarray } k \text{ (matrix-to-iarray } A) = \text{vec-to-iarray (column$
 $(\text{from-nat } k::'\text{cols}) A)$
by $(\text{metis } k \text{ vec-to-iarray-column'}$
show $((\text{column-iarray } k \text{ (matrix-to-iarray } A) \cdot i \text{ } x / (x \cdot i \text{ } x)) *_R x \text{ !! } i$
 $= (\text{column-iarray } k \text{ (matrix-to-iarray } A) \cdot i \text{ } x / (x \cdot i \text{ } x)) *_R x \text{ !! } i$
unfolding x
unfolding $\text{vec-to-iarray-column[symmetric, of from-nat } n::'\text{cols,$
 $\text{unfolded to-nat-from-nat-id[OF } n\text{-ncols[unfolded ncols-def]]}$
unfolding $c\text{-eq}$
unfolding $\text{vec-to-iarray-inner[symmetric]}$
unfolding $\text{vec-to-iarray-nth[OF } i\text{-nrows[unfolded nrows-def]]}$
unfolding $\text{vector-scaleR-component[symmetric]}$
unfolding $\text{vec-to-iarray-nth'[symmetric]}$
unfolding $\text{to-nat-from-nat-id[OF } i\text{-nrows[unfolded nrows-def]]}$
unfolding $\text{vec-to-iarray-scaleR ..}$

```

qed (simp)
  also have ... = (∑ x∈vec-to-iarray ‘ {column i A | i. i < from-nat k}.
(column-iarray k (matrix-to-iarray A) ·i x / (x ·i x)) *R x !! i)
  unfolding set-rw2[symmetric] ..
  also have ...=
    (∑ x∈{column i A | i. i < from-nat k}. (column (from-nat k) A · x / (x ·
x)) *R vec-to-iarray x !! i)
  unfolding sum.reindex[OF inj] o-def
  unfolding vec-to-iarray-column[of from-nat k::'cols,symmetric,unfolded
to-nat-from-nat-id[OF k[unfolded ncols-def]]]
  unfolding vec-to-iarray-inner[symmetric] ..
  also have ...
    = (∑ x∈{column i A | i. i < from-nat k}. column (from-nat k) A · x * x $
from-nat i / (x · x))
  unfolding vec-to-iarray-nth[OF i-nrows[unfolded nrows-def]] by auto
  finally have *: (∑ x∈(λn. column-iarray n (matrix-to-iarray A)) ‘ {0..<k}.
(column-iarray k (matrix-to-iarray A) ·i x / (x ·i x)) *R x) !! i
    = (∑ x∈{column i A | i. i < from-nat k}. column (from-nat k) A · x * x $
from-nat i / (x · x)) .
  have (column-iarray k (matrix-to-iarray A) –
    (∑ x∈(λn. column-iarray n (matrix-to-iarray A)) ‘ {0..<k}. (column-iarray
k (matrix-to-iarray A) ·i x / (x ·i x)) *R x) !! i =
    (column-iarray k (matrix-to-iarray A) !! i –
    (∑ x∈(λn. column-iarray n (matrix-to-iarray A)) ‘ {0..<k}. (column-iarray
k (matrix-to-iarray A) ·i x / (x ·i x)) *R x) !! i)
  proof (rule minus-iarray-component)
    have finite: finite ((λn. column-iarray n (matrix-to-iarray A)) ‘ {0..<k})
      using False by auto
    have not-empty: (λn. column-iarray n (matrix-to-iarray A)) ‘ {0..<k} ≠ {}
      by (metis False atLeastLessThan-empty-iff2 empty-is-image neq0-conv)
    let ?C={IArray.length ((column-iarray k (matrix-to-iarray A) ·i x / (x ·i
x)) *R x) | x. x ∈ (λn. column-iarray n (matrix-to-iarray A)) ‘ {0..<k}}
    have finite-C: finite ?C by auto
    have not-empty-C: ?C ≠ {} using False by auto
    let ?x=(column-iarray 0 (matrix-to-iarray A))
    let ?c=IArray.length ((column-iarray k (matrix-to-iarray A) ·i ?x / (?x ·i
?x)) *R ?x)
    show i < IArray.length (column-iarray k (matrix-to-iarray A))
      unfolding column-iarray-def
      by (auto, metis i-nrows-iarray IArray.length-def nrows-iarray-def)
    show i < IArray.length (∑ x∈(λn. column-iarray n (matrix-to-iarray A)) ‘
{0..<k}. (column-iarray k (matrix-to-iarray A) ·i x / (x ·i x)) *R x)
      unfolding length-sum-iarray[OF finite not-empty]
      unfolding Max-gr-iff[OF finite-C not-empty-C]
    proof (rule bexI[of - ?c])
      show i < ?c
    proof (unfold column-iarray-def, auto)
      have i < card (UNIV::'rows set)
        by (metis (no-types) i-nrows nrows-def)

```

thus $i < \text{length } (\text{IArray.list-of } ((\text{IArray } (\text{map } (\lambda n. \text{IArray.list-of } (\text{IArray.list-of } (\text{matrix-to-iarray } A) ! n) ! k) [0..<\text{length } (\text{IArray.list-of } (\text{matrix-to-iarray } A))])) \cdot i \text{ IArray } (\text{map } (\lambda n. \text{IArray.list-of } (\text{IArray.list-of } (\text{matrix-to-iarray } A) ! n) ! 0) [0..<\text{length } (\text{IArray.list-of } (\text{matrix-to-iarray } A))]) / (\text{IArray } (\text{map } (\lambda n. \text{IArray.list-of } (\text{IArray.list-of } (\text{matrix-to-iarray } A) ! n) ! 0) [0..<\text{length } (\text{IArray.list-of } (\text{matrix-to-iarray } A))])) \cdot i \text{ IArray } (\text{map } (\lambda n. \text{IArray.list-of } (\text{IArray.list-of } (\text{matrix-to-iarray } A) ! n) ! 0) [0..<\text{length } (\text{IArray.list-of } (\text{matrix-to-iarray } A))])))) *_R \text{IArray } (\text{map } (\lambda n. \text{IArray.list-of } (\text{IArray.list-of } (\text{matrix-to-iarray } A) ! n) ! 0) [0..<\text{length } (\text{IArray.list-of } (\text{matrix-to-iarray } A))]))))$

by (*simp add: matrix-to-iarray-def scaleR-iarray-def*)

qed

show $?c \in \{\text{IArray.length } ((\text{column-iarray } k (\text{matrix-to-iarray } A) \cdot i \ x / (x \cdot i \ x)) *_R \ x) \mid x \in (\lambda n. \text{column-iarray } n (\text{matrix-to-iarray } A)) \text{ ' } \{0..<k\}\}$

using *False by auto*

qed

qed

also have $\dots = \text{column-iarray } k (\text{matrix-to-iarray } A) !! i - (\sum_{x \in \{\text{column } i \ A \mid i. i < \text{from-nat } k\}. \text{column } (\text{from-nat } k) \ A \cdot x * x \ \$ \text{from-nat } i / (x \cdot x)} \text{column } (\text{from-nat } k) \ A \cdot x * x \ \$ \text{from-nat } i / (x \cdot x))$

unfolding $* \dots$

also have $\dots = \text{column } (\text{from-nat } k) \ A \ \$ \text{from-nat } i - (\sum_{x \in \{\text{column } i \ A \mid i. i < \text{from-nat } k\}. \text{column } (\text{from-nat } k) \ A \cdot x * x \ \$ \text{from-nat } i / (x \cdot x)} \text{column } (\text{from-nat } k) \ A \cdot x * x \ \$ \text{from-nat } i / (x \cdot x))$

unfolding *vec-to-iarray-column[of from-nat k::'cols,symmetric,unfolded to-nat-from-nat-id[OF k[unfolded ncols-def]]]*

unfolding *vec-to-iarray-nth[OF i-nrows[unfolded nrows-def]] ..*

finally show $\text{column } (\text{from-nat } k) \ A \ \$ \text{from-nat } i - (\sum_{x \in \{\text{column } i \ A \mid i. i < \text{from-nat } k\}. \text{column } (\text{from-nat } k) \ A \cdot x * x \ \$ \text{from-nat } i / (x \cdot x)} \text{column } (\text{from-nat } k) \ A \cdot x * x \ \$ \text{from-nat } i / (x \cdot x)) = (\text{column-iarray } k (\text{matrix-to-iarray } A) - (\sum_{x \in (\lambda n. \text{column-iarray } n (\text{matrix-to-iarray } A)) \text{ ' } \{0..<k\}. (\text{column-iarray } k (\text{matrix-to-iarray } A) \cdot i \ x / (x \cdot i \ x)) *_R \ x)} \text{column-iarray } k (\text{matrix-to-iarray } A) \cdot i \ x / (x \cdot i \ x)) *_R \ x) !! i$

\dots

qed

show $\text{column } (\text{from-nat } ia) \ A \ \$ \text{from-nat } i = \text{column-iarray } ia (\text{matrix-to-iarray } A) !! i$

unfolding *vec-to-iarray-nth[symmetric, OF i-nrows[unfolded nrows-def]]*

unfolding *vec-to-iarray-column'[symmetric, OF ia-ncols] ..*

assume *ia-not-k: ia \neq k*

and *eq: from-nat ia = (from-nat k::'cols)*

have *ia=k by (rule from-nat-eq-imp-eq[OF eq ia-ncols[unfolded ncols-def]] k[unfolded ncols-def])*

thus $\text{column } (\text{from-nat } k) \ A \ \$ \text{from-nat } i - (\sum_{x \in \{\text{column } i \ A \mid i. i < \text{from-nat } k\}. \text{column } (\text{from-nat } k) \ A \cdot x * x \ \$ \text{from-nat } i / (x \cdot x)} \text{column } (\text{from-nat } k) \ A \cdot x * x \ \$ \text{from-nat } i / (x \cdot x)) = \text{column-iarray } ia (\text{matrix-to-iarray } A) !! i$

using *ia-not-k by contradiction*

qed

qed

```
lemma matrix-to-iarray-Gram-Schmidt-upt-k:
  fixes A::real^cols::{mod-type}^rows::{mod-type}
  assumes k: k < ncols A
  shows matrix-to-iarray (Gram-Schmidt-upt-k A k) = Gram-Schmidt-upt-k-iarrays
    (matrix-to-iarray A) k
  using k
proof (induct k)
  case 0
  show ?case unfolding Gram-Schmidt-upt-k-def Gram-Schmidt-upt-k-iarrays-def
    by (simp add: matrix-to-iarray-Gram-Schmidt-column-k[OF 0.prem])
next
  case (Suc k)
  have k2: k < ncols (Gram-Schmidt-upt-k A k) using Suc.prem unfolding ncols-def
  by simp
  have k2': Suc k < ncols (Gram-Schmidt-upt-k A k) using Suc.prem unfolding
    ncols-def .
  have list-rw: [0..<Suc (Suc k)] = [0..<Suc k] @ [(Suc k)] by simp
  have hyp: matrix-to-iarray (Gram-Schmidt-upt-k A k) = Gram-Schmidt-upt-k-iarrays
    (matrix-to-iarray A) k
    by (metis Suc.hyps Suc.prem Suc-lessD)
  show matrix-to-iarray (Gram-Schmidt-upt-k A (Suc k)) = Gram-Schmidt-upt-k-iarrays
    (matrix-to-iarray A) (Suc k)
    unfolding Gram-Schmidt-upt-k-def Gram-Schmidt-upt-k-iarrays-def
    unfolding list-rw
    unfolding foldl-append
    unfolding foldl.simps
    unfolding Gram-Schmidt-upt-k-def[symmetric] Gram-Schmidt-upt-k-iarrays-def[symmetric]
    unfolding hyp[symmetric]
    by (rule matrix-to-iarray-Gram-Schmidt-column-k[OF k2])
qed
```

```
lemma matrix-to-iarray-Gram-Schmidt-matrix[code-unfold]:
  fixes A::real^cols::{mod-type}^rows::{mod-type}
  shows matrix-to-iarray (Gram-Schmidt-matrix A) = Gram-Schmidt-matrix-iarrays
    (matrix-to-iarray A)
  unfolding Gram-Schmidt-matrix-def Gram-Schmidt-matrix-iarrays-def
  unfolding matrix-to-iarray-ncols[symmetric]
  by (rule matrix-to-iarray-Gram-Schmidt-upt-k, simp add: ncols-def)
```

Examples:

```
value let A = list-of-list-to-matrix [[4,5],[8,1],[-1,5]]::real^2^3
  in iarray-of-iarray-to-list-of-list (matrix-to-iarray (Gram-Schmidt-matrix A))
```

```
value let A = IArray[IArray[4,5],IArray[8,1],IArray[-1,5]]
  in iarray-of-iarray-to-list-of-list (Gram-Schmidt-matrix-iarrays A)
```

end

11 QR Decomposition over iarrays

```
theory QR-Decomposition-IArrays
imports
  Gram-Schmidt-IArrays
begin
```

11.1 QR Decomposition refinement over iarrays

definition *norm-iarray* $A = \text{sqrt } (A \cdot i \ A)$

definition *divide-by-norm-iarray* $A = \text{tabulate2 } (nrows\text{-iarray } A) (ncols\text{-iarray } A)$
 $(\lambda a \ b. ((1/norm\text{-iarray } (column\text{-iarray } b \ A)) *_R (column\text{-iarray } b \ A)) !! a)$

definition *QR-decomposition-iarrays* $A = (\text{let } Q = \text{divide-by-norm-iarray } (\text{Gram-Schmidt-matrix-iarrays } A)$
 $\text{in } (Q, \text{transpose-iarray } Q **i \ A))$

lemma *vec-to-iarray-norm*[code-unfold]:
shows $(norm \ A) = norm\text{-iarray } (vec\text{-to-iarray } A)$
unfolding *norm-eq-sqrt-inner norm-iarray-def*
unfolding *vec-to-iarray-inner ..*

lemma *matrix-to-iarray-divide-by-norm*[code-unfold]:
fixes $A::real^{cols::\{mod\text{-type}\} rows::\{mod\text{-type}\}}$
shows $matrix\text{-to-iarray } (divide\text{-by-norm } A) = divide\text{-by-norm-iarray } (matrix\text{-to-iarray } A)$

proof (*unfold iarray-exhaust2 list-eq-iff-nth-eq, rule conjI, auto, unfold IArray.sub-def[symmetric] IArray.length-def[symmetric]*)

show $IArray.length (matrix\text{-to-iarray } (divide\text{-by-norm } A)) = IArray.length (divide\text{-by-norm-iarray } (matrix\text{-to-iarray } A))$

unfolding *matrix-to-iarray-def divide-by-norm-iarray-def tabulate2-def* **unfolding** *nrows-iarray-def* **by** *auto*

fix i **assume** $i < IArray.length (matrix\text{-to-iarray } (divide\text{-by-norm } A))$

show $IArray.length (matrix\text{-to-iarray } (divide\text{-by-norm } A) !! i) = IArray.length (divide\text{-by-norm-iarray } (matrix\text{-to-iarray } A) !! i)$

unfolding *matrix-to-iarray-def divide-by-norm-iarray-def tabulate2-def*

unfolding *nrows-iarray-def ncols-iarray-def o-def*

proof –

have $f1: i < card (UNIV::'rows \text{ set})$ **by** (*metis i length-eq-card-rows*)

have $\bigwedge x_4. vec\text{-to-iarray } x_4 = IArray (map (\lambda uua. x_4 \$ (from\text{-nat } uua::'cols)::real) [0..<card (UNIV::'cols \text{ set})])$ **by** (*simp add: vec-to-iarray-def*)

hence $0 < card (UNIV::'rows \text{ set}) \wedge length (IArray.list\text{-of } (IArray (map (\lambda R. IArray.of\text{-fun } (\lambda Ra. ((1 / norm\text{-iarray } (column\text{-iarray } Ra (IArray (map (\lambda R.$

```

vec-to-iarray (A $ from-nat R) [0..<card (UNIV::'rows set))])) *R column-iarray
Ra (IArray (map (λR. vec-to-iarray (A $ from-nat R) [0..<card (UNIV::'rows
set))])) !! R) (card (UNIV::'cols set)) [0..<card (UNIV::'rows set)) !! i) = length
(IArray.list-of (IArray (map (λR. vec-to-iarray (divide-by-norm A $ from-nat R))
[0..<card (UNIV::'rows set))])) !! i) using f1 by auto
thus IArray.length (IArray (map (λx. vec-to-iarray (divide-by-norm A $ from-nat
x) [0..<card (UNIV::'rows set))])) !! i) = IArray.length (IArray.of-fun (λi. IAr-
ray.of-fun (λb. ((1 / norm-iarray (column-iarray b (IArray (map (λx. vec-to-iarray
(A $ from-nat x) [0..<card (UNIV::'rows set))])))) *R column-iarray b (IArray
(map (λx. vec-to-iarray (A $ from-nat x) [0..<card (UNIV::'rows set))])) !! i)
(IArray.length (IArray (map (λx. vec-to-iarray (A $ from-nat x) [0..<card (UNIV::'rows
set))])) !! 0))) (IArray.length (IArray (map (λx. vec-to-iarray (A $ from-nat x)
[0..<card (UNIV::'rows set))])) !! i) by (simp add: vec-to-iarray-def)
qed
fix ia assume ia: ia < IArray.length (matrix-to-iarray (divide-by-norm A) !! i)
have i-nrows: i < nrows A using i unfolding matrix-to-iarray-def nrows-def by
auto
have ia-ncols: ia < ncols A using ia unfolding matrix-to-iarray-def o-def vec-to-iarray-def
ncols-def
by (auto, metis (no-types) Ex-list-of-length i-nrows length-map list-of.simps
map-nth nrows-def nth-map)
have i-nrows-iarray: i < nrows-iarray (matrix-to-iarray A) using i-nrows by
(metis matrix-to-iarray-nrows)
have ia-ncols-iarray: ia < ncols-iarray (matrix-to-iarray A) using ia-ncols by
(metis matrix-to-iarray-ncols)
show matrix-to-iarray (divide-by-norm A) !! i !! ia
= divide-by-norm-iarray (matrix-to-iarray A) !! i !! ia
unfolding divide-by-norm-def divide-by-norm-iarray-def
unfolding matrix-to-iarray-nth[of - from-nat i::'rows from-nat ia::'cols,
unfolded to-nat-from-nat-id[OF i-nrows[unfolded nrows-def]]
to-nat-from-nat-id[OF ia-ncols[unfolded ncols-def]]]
unfolding tabulate2-def
unfolding of-fun-nth[OF i-nrows-iarray]
unfolding of-fun-nth[OF ia-ncols-iarray]
unfolding vec-to-iarray-column'[OF ia-ncols, symmetric]
unfolding vec-to-iarray-norm[symmetric]
unfolding vector-scaleR-component
unfolding vec-to-iarray-scaleR[symmetric]
unfolding vec-to-iarray-nth[OF i-nrows[unfolded nrows-def]]
unfolding normalize-def
by auto
qed

```

```

lemma matrix-to-iarray-fst-QR-decomposition[code-unfold]:
shows matrix-to-iarray (fst (QR-decomposition A)) = fst (QR-decomposition-iarrays
(matrix-to-iarray A))
proof (unfold iarray-exhaust2 list-eq-iff-nth-eq, rule conjI, auto, unfold IArray.sub-def[symmetric]
IArray.length-def[symmetric])

```

```

fix i ia
show IArray.length (matrix-to-iarray (fst (QR-decomposition A)))
  = IArray.length (fst (QR-decomposition-iarrays (matrix-to-iarray A)))
  and IArray.length (matrix-to-iarray (fst (QR-decomposition A)) !! i)
  = IArray.length (fst (QR-decomposition-iarrays (matrix-to-iarray A)) !! i)
  and matrix-to-iarray (fst (QR-decomposition A)) !! i !! ia
  = fst (QR-decomposition-iarrays (matrix-to-iarray A)) !! i !! ia
unfolding QR-decomposition-def QR-decomposition-iarrays-def Let-def fst-conv
unfolding matrix-to-iarray-divide-by-norm
unfolding matrix-to-iarray-Gram-Schmidt-matrix by rule+
qed

lemma matrix-to-iarray-snd-QR-decomposition[code-unfold]:
  shows matrix-to-iarray (snd (QR-decomposition A)) = snd (QR-decomposition-iarrays
(matrix-to-iarray A))
proof (unfold iarray-exhaust2 list-eq-iff-nth-eq, rule conjI, auto, unfold IArray.sub-def[symmetric]
IArray.length-def[symmetric])
  fix i ia
  show IArray.length (matrix-to-iarray (snd (QR-decomposition A)))
    = IArray.length (snd (QR-decomposition-iarrays (matrix-to-iarray A)))
    and IArray.length (matrix-to-iarray (snd (QR-decomposition A)) !! i)
    = IArray.length (snd (QR-decomposition-iarrays (matrix-to-iarray A)) !! i)
    and matrix-to-iarray (snd (QR-decomposition A)) !! i !! ia
    = snd (QR-decomposition-iarrays (matrix-to-iarray A)) !! i !! ia
  unfolding QR-decomposition-iarrays-def QR-decomposition-def Let-def snd-conv
  unfolding matrix-to-iarray-matrix-matrix-mult
  unfolding matrix-to-iarray-transpose
  unfolding matrix-to-iarray-divide-by-norm
  unfolding matrix-to-iarray-Gram-Schmidt-matrix by rule+
qed

definition matrix-to-iarray-pair X = (matrix-to-iarray (fst X), matrix-to-iarray
(snd X))

lemma matrix-to-iarray-QR-decomposition[code-unfold]:
  shows matrix-to-iarray-pair (QR-decomposition A) = QR-decomposition-iarrays
(matrix-to-iarray A)
  unfolding matrix-to-iarray-pair-def
  unfolding matrix-to-iarray-fst-QR-decomposition
  unfolding matrix-to-iarray-snd-QR-decomposition by simp
end

```

12 Examples of execution using floats and IArrays

```

theory Examples-QR-IArrays-Float
imports
  QR-Decomposition-IArrays
  Gauss-Jordan.Code-Real-Approx-By-Float-Haskell

```

begin

12.1 Examples

definition *example1* = (let A = list-of-list-to-matrix [[1,2,4],[9,4,5],[0,0,0]]::real³³
in
iarray-of-iarray-to-list-of-list (matrix-to-iarray (divide-by-norm A)))

definition *example2* = (let A = list-of-list-to-matrix [[1,2,4],[9,4,5],[0,0,4]]::real³³
in
iarray-of-iarray-to-list-of-list (matrix-to-iarray (fst (QR-decomposition A))))

definition *example3* = (let A = list-of-list-to-matrix [[1,2,4],[9,4,5],[0,0,4]]::real³³
in
iarray-of-iarray-to-list-of-list (matrix-to-iarray (snd (QR-decomposition A))))

definition *example4* = (let A = list-of-list-to-matrix [[1,2,4],[9,4,5],[0,0,4]]::real³³
in
iarray-of-iarray-to-list-of-list (matrix-to-iarray (fst (QR-decomposition A)) ** (snd
(QR-decomposition A))))

definition *example5* = (let A = list-of-list-to-matrix [[1,sqrt 2,4],[sqrt 5,4,5],[0,sqrt
7,4]]::real³³ in
iarray-of-iarray-to-list-of-list (matrix-to-iarray (fst (QR-decomposition A))))

definition *example6* = (let A = list-of-list-to-matrix [[1,sqrt 2,4],[sqrt 5,4,5],[0,sqrt
7,4]]::real³³ in
iarray-of-iarray-to-list-of-list (matrix-to-iarray ((fst (QR-decomposition A))))

definition *example1b* = (let A = IArray[IArray[1,2,4],IArray[9,4,5::real],IArray[0,0,0]]
in
iarray-of-iarray-to-list-of-list ((divide-by-norm-iarray A)))

definition *example2b* = (let A = IArray[IArray[1,2,4],IArray[9,4,5],IArray[0,0,4]] in
iarray-of-iarray-to-list-of-list ((fst (QR-decomposition-iarrays A))))

definition *example3b* = (let A = IArray[IArray[1,2,4],IArray[9,4,5],IArray[0,0,4]]
in
iarray-of-iarray-to-list-of-list ((snd (QR-decomposition-iarrays A))))

definition *example4b* = (let A = IArray[IArray[1,2,4],IArray[9,4,5],IArray[0,0,4]]
in
iarray-of-iarray-to-list-of-list (
((fst (QR-decomposition-iarrays A)) **i (snd (QR-decomposition-iarrays A))))

definition *example5b* = (let A = IArray[IArray[1,2,4],IArray[9,4,5],IArray[0,0,4],IArray[3,5,4]] in
iarray-of-iarray-to-list-of-list (


```
((fst (QR-decomposition-iarrays A)) **i (snd (QR-decomposition-iarrays A))))
```

```
definition example6b = (let A = IArray [IArray [1,sqrt 2,4],IArray[sqrt 5,4,5],IArray[0,sqrt 7,4]]
  in iarray-of-iarray-to-list-of-list (fst (QR-decomposition-iarrays A)))
```

The following example is presented in Chapter 1 of the book *Numerical Methods in Scientific Computing* by Dahlquist and Bjorck

```
definition book-example = (let A = list-of-list-to-matrix
  [[1,-0.6691],[1,-0.3907],[1,-0.1219],[1,0.3090],[1,0.5878]]::real^2^5;
  b = list-to-vec [0.3704,0.5,0.6211,0.8333,0.9804]::real^5;
  QR = (QR-decomposition A);
  Q = fst QR;
  R = snd QR
  in IArray.list-of (vec-to-iarray (the (inverse-matrix R) ** transpose Q *v b)))
```

```
export-code example1 example2 example3 example4 example5 example6
  example1b example2b example3b example4b example5b example6b
book-example
  in SML module-name QR
```

```
end
```

13 Examples of execution using symbolic computation and iarrays

```
theory Examples-QR-IArrays-Symbolic
imports
  Examples-QR-Abstract-Symbolic
  QR-Decomposition-IArrays
begin
```

13.1 Execution of the QR decomposition using symbolic computation and iarrays

```
definition show-vec-real-iarrays v = IArray.of-fun (λi. show-real (v !! i)) (IArray.length v)
```

```
lemma vec-to-iarray-show-vec-real[code-unfold]: vec-to-iarray (show-vec-real v)
  = show-vec-real-iarrays (vec-to-iarray v)
unfolding show-vec-real-def show-vec-real-iarrays-def vec-to-iarray-def by auto
```

The following function is used to print elements of type `vec` as lists of characters; useful for printing vectors in the output panel.

```
definition print-vec = IArray.list-of ∘ show-vec-real-iarrays ∘ vec-to-iarray
```

```
definition show-matrix-real-iarrays A = IArray.of-fun (λi. show-vec-real-iarrays (A !! i)) (IArray.length A)
```

lemma *matrix-to-iarray-show-matrix-real*[code-unfold]: *matrix-to-iarray* (*show-matrix-real* *v*)
= *show-matrix-real-iarrays* (*matrix-to-iarray* *v*)
unfolding *show-matrix-real-iarrays-def show-matrix-real-def*
unfolding *matrix-to-iarray-def*
by (*simp add: vec-to-iarray-show-vec-real*)

The following functions are useful to print matrices as lists of lists of characters; useful for printing in the output panel.

definition *print-vec-mat* = *IArray.list-of* \circ *show-vec-real-iarrays*

definition *print-mat-aux* *A* = *IArray.of-fun* ($\lambda i. \text{print-vec-mat } (A !! i)$) (*IArray.length* *A*)

definition *print-mat* = *IArray.list-of* \circ *print-mat-aux* \circ *matrix-to-iarray*

13.1.1 Examples

value *let* *A* = *list-of-list-to-matrix* $[[1,2,4],[9,4,5],[0,0,0]]::\text{real}^3^3$ *in*
iarray-of-iarray-to-list-of-list (*matrix-to-iarray* (*show-matrix-real* (*divide-by-norm* *A*))))

value *let* *A* = *list-of-list-to-matrix* $[[1,2,4],[9,4,5],[0,0,4]]::\text{real}^3^3$ *in*
iarray-of-iarray-to-list-of-list (*matrix-to-iarray* (*show-matrix-real* (*fst* (*QR-decomposition* *A*))))

value *let* *A* = *list-of-list-to-matrix* $[[1,2,4],[9,4,5],[0,0,4]]::\text{real}^3^3$ *in*
iarray-of-iarray-to-list-of-list (*matrix-to-iarray* (*show-matrix-real* (*snd* (*QR-decomposition* *A*))))

value *let* *A* = *list-of-list-to-matrix* $[[1,2,4],[9,4,5],[0,0,4]]::\text{real}^3^3$ *in*
iarray-of-iarray-to-list-of-list (*matrix-to-iarray*
(*show-matrix-real* ((*fst* (*QR-decomposition* *A*)) ** (*snd* (*QR-decomposition* *A*))))))

value *let* *A* = *list-of-list-to-matrix* $[[1,2,3],[9,4,5],[0,0,4],[1,2,3]]::\text{real}^3^4$ *in* *rank* *A* = *ncols* *A*

value *let* *A* = *list-of-list-to-matrix* $[[1,2,3],[9,4,5],[0,0,4],[1,2,3]]::\text{real}^3^4$;
b = *list-to-vec* $[1,2,3,4]::\text{real}^4$ *in*
print-result-solve (*solve* *A* *b*)

value *let* *A* = *list-of-list-to-matrix* $[[1,2,3],[9,4,5],[0,0,4],[1,2,3]]::\text{real}^3^4$;
b = *list-to-vec* $[1,2,3,4]::\text{real}^4$
in
vec-to-list (*show-vec-real* (*the* (*inverse-matrix* (*snd* (*QR-decomposition* *A*)))) **
transpose (*fst* (*QR-decomposition* *A*)) * *v* *b*)

value let $A = \text{list-of-list-to-matrix } [[1,2,3],[9,4,5],[0,0,4],[1,2,3]]::\text{real}^3^4$;
 $b = \text{list-to-vec } [1,2,3,4]::\text{real}^4$
in $\text{matrix-to-list-of-list } (\text{show-matrix-real } ((\text{snd } (\text{QR-decomposition } A))))$

least squares solution

definition $A \equiv \text{list-of-list-to-matrix } [[1,3/5,3],[9,4,5/3],[0,0,4],[1,2,3]]::\text{real}^3^4$

definition $b \equiv \text{list-to-vec } [1,2,3,4]::\text{real}^4$

value let $Q = \text{fst } (\text{QR-decomposition } A)$; $R = \text{snd } (\text{QR-decomposition } A)$
in $\text{print-vec } ((\text{the } (\text{inverse-matrix } R) ** \text{transpose } Q * v b))$

A times least squares solution

value let $Q = \text{fst } (\text{QR-decomposition } A)$; $R = \text{snd } (\text{QR-decomposition } A)$
in $\text{print-vec } (A * v (\text{the } (\text{inverse-matrix } R) ** \text{transpose } Q * v b))$

The matrix Q

value $\text{print-mat } (\text{fst } (\text{QR-decomposition } A))$

The matrix R

value $\text{print-mat } (\text{snd } (\text{QR-decomposition } A))$

The inverse of matrix R

value let $R = \text{snd } (\text{QR-decomposition } A)$ in $\text{print-mat } (\text{the } (\text{inverse-matrix } R))$

The least squares solution is in the left null space of A

value let $Q = \text{fst } (\text{QR-decomposition } A)$; $R = \text{snd } (\text{QR-decomposition } A)$;
 $b2 = (A * v (\text{the } (\text{inverse-matrix } R) ** \text{transpose } Q * v b))$
in $\text{print-vec } ((b - b2)v * A)$

value let $A = \text{list-of-list-to-matrix } [[1,2,4],[9,4,5],[0,0,4],[3,5,4]]::\text{real}^3^4$ in
 $\text{iarray-of-iarray-to-list-of-list } (\text{matrix-to-iarray } (\text{show-matrix-real } ((\text{fst } (\text{QR-decomposition } A)) ** (\text{snd } (\text{QR-decomposition } A))))))$

value let $A = \text{IArray}[\text{IArray}[1,2,4],\text{IArray}[9,4,5::\text{real}],\text{IArray}[0,0,0]]$ in
 $\text{iarray-of-iarray-to-list-of-list } (\text{show-matrix-real-iarrays } (\text{divide-by-norm-iarray } A))$

value let $A = \text{IArray}[\text{IArray}[1,2,4],\text{IArray}[9,4,5],\text{IArray}[0,0,4]]$ in
 $\text{iarray-of-iarray-to-list-of-list } (\text{show-matrix-real-iarrays } (\text{fst } (\text{QR-decomposition-iarrays } A)))$

value let $A = \text{IArray}[\text{IArray}[1,2,4],\text{IArray}[9,4,5],\text{IArray}[0,0,4]]$ in
 $\text{iarray-of-iarray-to-list-of-list } (\text{show-matrix-real-iarrays } (\text{snd } (\text{QR-decomposition-iarrays } A)))$

value let $A = \text{list-of-list-to-matrix } [[1,2,3],[9,4,5],[0,0,4],[1,2,3]]::\text{real}^3^4$ in rank
 $A = \text{ncols } A$

```
value let A = list-of-list-to-matrix [[1,2,3],[9,4,5],[0,0,4],[1,2,3]]::real^3^4;
      b = list-to-vec [1,2,3,4]::real^4 in
      print-result-solve (solve A b)
```

```
value let A = list-of-list-to-matrix [[1,2,3],[9,4,5],[0,0,4],[1,2,3]]::real^3^4;
      b = list-to-vec [1,2,3,4]::real^4
      in
      vec-to-list (show-vec-real (the (inverse-matrix (snd (QR-decomposition A))) **
transpose (fst (QR-decomposition A)) *v b))
```

```
value let A = list-of-list-to-matrix [[1,2,3],[9,4,5],[0,0,4],[1,2,3]]::real^3^4;
      b = list-to-vec [1,2,3,4]::real^4
      in matrix-to-list-of-list (show-matrix-real ((snd (QR-decomposition A))))
```

```
value let A = list-of-list-to-matrix [[1,2,3],[9,4,5],[0,0,4],[1,2,3]]::real^3^4;
      b = list-to-vec [1,2,3,4]::real^4;
      b2 = (A *v (the (inverse-matrix (snd (QR-decomposition A))) ** transpose (fst
(QR-decomposition A)) *v b))
      in
      vec-to-list (show-vec-real ((b - b2)v* A))
```

```
value let A = IArray[IArray[1,2,4],IArray[9,4,5],IArray[0,0,4]] in
      iarray-of-iarray-to-list-of-list (show-matrix-real-iarrays
      ((fst (QR-decomposition-iarrays A)) **i (snd (QR-decomposition-iarrays A))))
```

```
value let A = IArray[IArray[1,2,4],IArray[9,4,5],IArray[0,0,4],IArray[3,5,4]] in
      iarray-of-iarray-to-list-of-list (show-matrix-real-iarrays
      ((fst (QR-decomposition-iarrays A)) **i (snd (QR-decomposition-iarrays A))))
```

The following example is presented in Chapter 1 of the book *Numerical Methods in Scientific Computing* by Dahlquist and Bjorck

```
value let A = list-of-list-to-matrix
[[1,-0.6691],[1,-0.3907],[1,-0.1219],[1,0.3090],[1,0.5878]]::real^2^5;
      b = list-to-vec [0.3704,0.5,0.6211,0.8333,0.9804]::real^5;
      QR = (QR-decomposition A);
      Q = fst QR;
      R = snd QR
      in print-vec (the (inverse-matrix R) ** transpose Q *v b)
```

```
definition example = (let A = IArray[IArray[1,2,4],IArray[9,4,5],IArray[0,0,4],IArray[3,5,4]] in
      iarray-of-iarray-to-list-of-list (show-matrix-real-iarrays
      ((fst (QR-decomposition-iarrays A)) **i (snd (QR-decomposition-iarrays A))))
```

`export-code example in SML module-name QR`

`end`

14 Generalization of the Second Part of the Fundamental Theorem of Linear Algebra

`theory Generalizations2`

`imports`

`Rank-Nullity-Theorem.Fundamental-Subspaces`

`begin`

14.1 Conjugate class

`class cnj = field +`

`fixes cnj :: 'a ⇒ 'a`

`assumes cnj-idem[simp]: cnj (cnj a) = a`

`and cnj-add: cnj (a+b) = cnj a + cnj b`

`and cnj-mult: cnj (a * b) = cnj a * cnj b`

`begin`

`lemma two-not-one: 2 ≠ (1::'a)`

`proof (rule ccontr, simp)`

`assume 2 = (1::'a)`

`hence 2 - 1 = 1 - (1::'a) by auto`

`hence 1 = (0::'a) by auto`

`thus False using one-neq-zero by contradiction`

`qed`

`lemma cnj-0[simp]: cnj 0 = 0`

`proof -`

`have cnj 0 = cnj (0 + 0) by auto`

`also have ... = cnj 0 + cnj 0 unfolding cnj-add ..`

`also have ... = 2 * (cnj 0) by (simp add: local.mult-2)`

`finally have cnj 0 = 2 * cnj 0 .`

`thus ?thesis by (auto simp add: two-not-one)`

`qed`

`lemma cnj-0-eq[simp]: (cnj a = 0) = (a = 0)`

`proof auto`

`assume cnj-rw: 0 = cnj a`

`have cnj (cnj a) = a using cnj-idem by simp`

`hence cnj 0 = a unfolding cnj-rw .`

`hence a = 0 by simp`

`thus a = cnj a using cnj-rw by simp`

`qed`

`lemma a-cnj-a-0: (a * cnj a = 0) = (a = 0)`

```

    by simp
end

lemma cnj-sum: cnj ( $\sum_{xa \in A} ((f xa))$ ) = ( $\sum_{xa \in A} cnj (f xa)$ )
  by (cases finite A, induct set: finite, auto simp add: cnj-add)

instantiation real :: cnj
begin

definition (cnj-real :: real  $\Rightarrow$  real) = id

instance
by (intro-classes, auto simp add: cnj-real-def)
end

instantiation complex :: cnj
begin

definition (cnj-complex :: complex  $\Rightarrow$  complex) = Complex.cnj

instance
by (intro-classes, auto simp add: cnj-complex-def)
end

```

14.2 Real_of_extended class

```

class real-of-extended = real-vector + cnj +
  fixes real-of :: 'a  $\Rightarrow$  real
  assumes real-add: real-of ((a::'a) + b) = real-of a + real-of b
  and real-uminus: real-of (-a) = - real-of a
  and real-scalar-mult: real-of (c *R a) = c * (real-of a)
  and real-a-cnj-ge-0: real-of (a * cnj a)  $\geq$  0
begin

lemma real-minus: real-of (a - b) = real-of a - real-of b
proof -
  have real-of (a - b) = real-of (a + - b) by simp
  also have ... = real-of a + real-of (- b) unfolding real-add ..
  also have ... = real-of a + - real-of b unfolding real-uminus ..
  also have ... = real-of a - real-of b by simp
  finally show ?thesis .
qed

lemma real-0[simp]: real-of 0 = 0
proof -
  have real-of 0 = real-of (0+0) by auto
  also have ... = real-of 0 + real-of 0 unfolding real-add ..

```

```

    also have ... = 2*(real-of 0) by auto
    finally have real-of 0 = 2* real-of 0 .
    thus ?thesis by (auto simp add: two-not-one)
qed

```

```

lemma real-sum:
  real-of (sum ( $\lambda i. f i$ ) A) = sum ( $\lambda i. real-of (f i)$ ) A
proof (cases finite A)
  case False thus ?thesis by auto
next
  case True
  thus ?thesis
  by (induct, auto simp add: real-add)
qed

```

end

```

instantiation real :: real-of-extended
begin

```

```

definition real-of-real :: real  $\Rightarrow$  real where real-of-real = id

```

```

instance
  by (intro-classes, auto simp add: real-of-real-def cnj-real-def)
end

```

```

instantiation complex :: real-of-extended
begin

```

```

definition real-of-complex :: complex  $\Rightarrow$  real where real-of-complex = Re

```

```

instance
  by (intro-classes, auto simp add: real-of-complex-def cnj-complex-def)
end

```

14.3 Generalizing HMA

14.3.1 Inner product spaces

We generalize the *real-inner class* to more general inner product spaces.

```

locale inner-product-space = vector-space scale
  for scale :: ('a::{field, cnj, real-of-extended} => 'b::ab-group-add => 'b) +
  fixes inner :: 'b  $\Rightarrow$  'b  $\Rightarrow$  'a
  assumes inner-commute: inner x y = cnj (inner y x)
  and inner-add-left: inner (x+y) z = inner x z + inner y z
  and inner-scaleR-left [simp]:inner (scale r x) y = r * inner x y
  and inner-ge-zero [simp]:0  $\leq$  real-of (inner x x)
  and inner-eq-zero-iff [simp]: inner x x = 0  $\longleftrightarrow$  x=0

```

and *real-scalar-mult2*: *real-of (inner x x) *_R A = inner x x * A*
and *inner-gt-zero-iff*: $0 < \text{real-of } (inner\ x\ x) \iff x \neq 0$

interpretation *RV-inner: inner-product-space scaleR inner*

by (*unfold-locales*) (*auto simp: cnj-real-def inner-add-left real-of-real-def algebra-simps inner-commute*)

interpretation *RR-inner: inner-product-space scaleR (*)*

by (*unfold-locales, auto simp add: cnj-real-def distrib-right real-of-real-def*)
(*metis not-real-square-gt-zero*)

interpretation *CC-inner: inner-product-space ((*): complex \Rightarrow complex \Rightarrow complex)*

$\lambda x\ y. x * cnj\ y$

apply (*unfold-locales*)

apply (*auto simp add: real-of-complex-def cnj-complex-def distrib-left distrib-right complex-mult-cnj complex-neq-0 cmod-power2 complex-norm-square*)

apply (*metis Re-complex-of-real complex-neq-0 less-numeral-extra(3) of-real-add of-real-power zero-complex.simps(1)*)

by (*simp add: distrib-left mult.commute scaleR-conv-of-real*)

context *inner-product-space*

begin

lemma *inner-zero-left [simp]: inner 0 x = 0*

using *inner-add-left [of 0 0 x]* **by** (*auto simp add: two-not-one*)

lemma *inner-minus-left [simp]: inner (- x) y = - inner x y*

using *inner-add-left [of x - x y]* **using** *add-eq-0-iff* **by** *force*

lemma *inner-diff-left: inner (x - y) z = inner x z - inner y z*

using *inner-add-left [of x - y z]* **by** *simp*

lemma *inner-sum-left: inner ($\sum_{x \in A} f\ x$) y = ($\sum_{x \in A} \text{inner } (f\ x)\ y$)*

by (*cases finite A, induct set: finite, simp-all add: inner-add-left*)

Transfer distributivity rules to right argument.

lemma *inner-add-right: inner x (y + z) = inner x y + inner x z*

proof -

have *inner x (y + z) = cnj (inner (y + z) x)* **using** *inner-commute* **by** *blast*

also have $\dots = cnj ((inner\ y\ x) + (inner\ z\ x))$ **using** *inner-add-left* **by** *simp*

also have $\dots = cnj (inner\ y\ x) + cnj (inner\ z\ x)$ **using** *cnj-add* **by** *blast*

also have $\dots = inner\ x\ y + inner\ x\ z$

using *inner-commute[of x y]* **using** *inner-commute[of x z]* **by** *simp*

finally show *?thesis* .

qed

lemma *inner-scaleR-right* [simp]: $inner\ x\ (scale\ r\ y) = (cnj\ r) * (inner\ x\ y)$
using *inner-scaleR-left* [of $r\ y\ x$]
by (metis (no-types) *cnj-mult local.inner-commute*)

lemma *inner-zero-right* [simp]: $inner\ x\ 0 = 0$
using *inner-zero-left* [of x]
by (metis *local.inner-commute local.inner-eq-zero-iff*)

lemma *inner-minus-right* [simp]: $inner\ x\ (-\ y) = -\ inner\ x\ y$
using *inner-minus-left* [of $y\ x$]
by (metis (no-types) *add-eq-0-iff local.inner-add-right local.inner-zero-right*)

lemma *inner-diff-right*: $inner\ x\ (y - z) = inner\ x\ y - inner\ x\ z$
using *inner-diff-left* [of $y\ z\ x$]
by (metis *add-uminus-conv-diff local.inner-add-right local.inner-minus-right*)

lemma *inner-sum-right*: $inner\ x\ (\sum_{y \in A} f\ y) = (\sum_{y \in A} inner\ x\ (f\ y))$
proof –
have $inner\ x\ (\sum_{y \in A} f\ y) = cnj\ (inner\ (\sum_{y \in A} f\ y)\ x)$ **using** *inner-commute*
by *blast*
also have $\dots = cnj\ (\sum_{xa \in A} inner\ (f\ xa)\ x)$ **unfolding** *inner-sum-left* [of $f\ A\ x$] ..
also have $\dots = (\sum_{xa \in A} cnj\ (inner\ (f\ xa)\ x))$ **unfolding** *cnj-sum* ..
also have $\dots = (\sum_{xa \in A} inner\ x\ (f\ xa))$ **by** (rule *sum.cong, simp, metis inner-commute*)
finally show ?thesis .
qed

lemmas *inner-add* [algebra-simps] = *inner-add-left inner-add-right*
lemmas *inner-diff* [algebra-simps] = *inner-diff-left inner-diff-right*
lemmas *inner-scaleR* = *inner-scaleR-left inner-scaleR-right*

Legacy theorem names

lemmas *inner-left-distrib* = *inner-add-left*
lemmas *inner-right-distrib* = *inner-add-right*
lemmas *inner-distrib* = *inner-left-distrib inner-right-distrib*

lemma *aux-Cauchy*:
shows $0 \leq real-of\ (inner\ x\ x + (cnj\ a) * (inner\ x\ y) + a * ((cnj\ (inner\ x\ y)) + (cnj\ a) * (inner\ y\ y)))$
proof –
have $(inner\ (x+scale\ a\ y)\ (x+scale\ a\ y)) = (inner\ (x+scale\ a\ y)\ x) + (inner\ (x+scale\ a\ y)\ (scale\ a\ y))$
unfolding *inner-add-right* ..

also have $\dots = \text{inner } x x + (\text{cnj } a) * (\text{inner } x y) + a * ((\text{cnj } (\text{inner } x y)) + (\text{cnj } a) * (\text{inner } y y))$
unfolding *inner-add-left*
unfolding *inner-scaleR-left*
unfolding *inner-scaleR-right*
unfolding *inner-commute[of y x]*
unfolding *distrib-left*
by *auto*
finally show *?thesis* **by** (*metis inner-ge-zero*)
qed

lemma *real-inner-inner*: $\text{real-of } (\text{inner } x x * \text{inner } y y) = \text{real-of } (\text{inner } x x) * \text{real-of } (\text{inner } y y)$
by (*metis real-scalar-mult real-scalar-mult2*)

lemma *Cauchy-Schwarz-ineq*:

$\text{real-of } (\text{cnj } (\text{inner } x y) * \text{inner } x y) \leq \text{real-of } (\text{inner } x x) * \text{real-of } (\text{inner } y y)$

proof –

define *cnj-a* **where** $\text{cnj-a} = - \text{cnj } (\text{inner } x y) / (\text{inner } y y)$

define *a* **where** $a = \text{cnj } (\text{cnj-a})$

have *cnj-rw*: $(\text{cnj } a) = \text{cnj-a}$

unfolding *a-def* **by** (*simp*)

have *rw-0*: $(\text{cnj } (\text{inner } x y)) + (\text{cnj } a) * (\text{inner } y y) = 0$

unfolding *cnj-rw cnj-a-def* **by** *auto*

have $0 \leq \text{real-of } (\text{inner } x x + (\text{cnj } a) * (\text{inner } x y) + a * ((\text{cnj } (\text{inner } x y)) + (\text{cnj } a) * (\text{inner } y y)))$

using *aux-Cauchy* **by** *auto*

also have $\dots = \text{real-of } (\text{inner } x x + (\text{cnj } a) * (\text{inner } x y))$ **unfolding** *rw-0* **by** *auto*

also have $\dots = \text{real-of } (\text{inner } x x - \text{cnj } (\text{inner } x y) * \text{inner } x y / \text{inner } y y)$

unfolding *cnj-rw cnj-a-def* **by** *auto*

finally have $0 \leq \text{real-of } (\text{inner } x x - \text{cnj } (\text{inner } x y) * \text{inner } x y / \text{inner } y y)$

hence $0 \leq \text{real-of } (\text{inner } y y) * \text{real-of } (\text{inner } x x - \text{cnj } (\text{inner } x y) * \text{inner } x y / \text{inner } y y)$ **by** *auto*

also have $\dots = \text{real-of } (\text{real-of } (\text{inner } y y) *_{\mathbb{R}} (\text{inner } x x - \text{cnj } (\text{inner } x y) * \text{inner } x y / \text{inner } y y))$

unfolding *real-scalar-mult ..*

also have $\dots = \text{real-of } ((\text{inner } y y) * (\text{inner } x x - \text{cnj } (\text{inner } x y) * \text{inner } x y / \text{inner } y y))$

unfolding *real-scalar-mult2 ..*

also have $\dots = \text{real-of } ((\text{inner } x x - \text{cnj } (\text{inner } x y) * \text{inner } x y / \text{inner } y y) * (\text{inner } y y))$

by (*simp add: mult.commute*)

also have $\dots = \text{real-of } (((\text{inner } x x) * (\text{inner } y y) - \text{cnj } (\text{inner } x y) * \text{inner } x y))$

by (*simp add: left-diff-distrib*)

also have $\dots = \text{real-of } ((\text{inner } x x) * (\text{inner } y y)) - \text{real-of } (\text{cnj } (\text{inner } x y) * \text{inner } x y)$

unfolding *real-minus ..*

finally have $\text{real-of } (\text{cnj } (\text{inner } x \ y) * \text{inner } x \ y) \leq \text{real-of } ((\text{inner } x \ x) * (\text{inner } y \ y))$ **by** *auto*
thus *?thesis* **unfolding** *real-inner-inner* .
qed
end

hide-const (open) *norm*

context *inner-product-space*
begin

definition $\text{norm } x = (\text{sqrt } (\text{real-of } (\text{inner } x \ x)))$

lemmas *norm-eq-sqrt-inner = norm-def*

lemma *inner-cnj-ge-zero[simp]*: $\text{real-of } ((\text{inner } x \ y) * \text{cnj } (\text{inner } x \ y)) \geq 0$
using *real-a-cnj-ge-0* **by** *auto*

lemma *power2-norm-eq-inner*: $(\text{norm } x)^2 = \text{real-of } (\text{inner } x \ x)$
by (*simp add: norm-def*)

lemma *Cauchy-Schwarz-ineq2*:

$\text{sqrt } (\text{real-of } (\text{cnj } (\text{inner } x \ y) * \text{inner } x \ y)) \leq \text{norm } x * \text{norm } y$

proof (*rule power2-le-imp-le*)

have $0 \leq \text{real-of } (\text{cnj } (\text{inner } x \ y) * \text{inner } x \ y)$

by (*simp add: mult.commute*)

have $\text{real-of } (\text{cnj } (\text{inner } x \ y) * \text{inner } x \ y) \leq \text{real-of } (\text{inner } x \ x) * \text{real-of } (\text{inner } y \ y)$

using *Cauchy-Schwarz-ineq* .

thus $(\text{sqrt } (\text{real-of } (\text{cnj } (\text{inner } x \ y) * \text{inner } x \ y))))^2 \leq (\text{norm } x * \text{norm } y)^2$

unfolding *power-mult-distrib*

unfolding *power2-norm-eq-inner* **unfolding** *real-sqrt-pow2[OF eq]* .

show $0 \leq \text{norm } x * \text{norm } y$

unfolding *norm-eq-sqrt-inner*

by (*intro mult-nonneg-nonneg real-sqrt-ge-zero inner-ge-zero*)

qed

end

14.3.2 Orthogonality

hide-const (open) *orthogonal*

context *inner-product-space*
begin

definition *orthogonal* $x \ y \longleftrightarrow \text{inner } x \ y = 0$

lemma *orthogonal-clauses*:

orthogonal a 0
orthogonal a x \implies orthogonal a (scale c x)
orthogonal a x \implies orthogonal a (- x)
orthogonal a x \implies orthogonal a y \implies orthogonal a (x + y)
orthogonal a x \implies orthogonal a y \implies orthogonal a (x - y)
orthogonal 0 a
orthogonal x a \implies orthogonal (scale c x) a
orthogonal x a \implies orthogonal (- x) a
orthogonal x a \implies orthogonal y a \implies orthogonal (x + y) a
orthogonal x a \implies orthogonal y a \implies orthogonal (x - y) a
unfolding *orthogonal-def inner-add inner-diff* **by** *auto*

lemma *inner-commute-zero: (inner xa x = 0) = (inner x xa = 0)*
by *(metis cnj-0 local.inner-commute)*

lemma *vector-sub-project-orthogonal:*

inner b (x - scale (inner x b / (inner b b)) b) = 0

proof -

have *f1: $\bigwedge b a ba. inner b (scale a ba) = cnj (a * inner ba b)$*

by *(metis local.inner-commute local.inner-scaleR-left)*

{ assume *b \neq 0*

hence *cnj (inner x b) = inner b x \wedge inner b b \neq 0*

by *(metis (no-types) local.inner-commute local.inner-eq-zero-iff)*

hence *inner b (x - scale (inner x b / inner b b) b) = 0*

using *f1 local.inner-diff-right* **by** *force* **}**

thus *?thesis* **by** *fastforce*

qed

lemma *orthogonal-commute: orthogonal x y \longleftrightarrow orthogonal y x*

unfolding *orthogonal-def* **using** *inner-commute-zero* **by** *auto*

lemma *pairwise-orthogonal-insert:*

assumes *pairwise orthogonal S*

and $\bigwedge y. y \in S \implies$ *orthogonal x y*

shows *pairwise orthogonal (insert x S)*

using *assms* **unfolding** *pairwise-def*

by *(auto simp add: orthogonal-commute)*

end

lemma *sum-0-all:*

assumes *a: $\forall a \in A. f a \geq (0 :: real)$*

and *s0: sum f A = 0* **and** *f: finite A*

shows $\forall a \in A. f a = 0$

using *a f s0 sum-nonneg-eq-0-iff* **by** *blast*

14.4 Vecs as inner product spaces

locale *vec-real-inner = F?: inner-product-space ((* :: 'a \Rightarrow 'a \Rightarrow 'a) inner-field*

for *inner-field* :: 'a \Rightarrow 'a \Rightarrow 'a::{*field,cnj,real-of-extended*}
+ **fixes** *inner* :: 'a \wedge 'n \Rightarrow 'a \wedge 'n \Rightarrow 'a
assumes *inner-vec-def*: *inner* *x* *y* = *sum* (λ i. *inner-field* (*x*\$*i*) (*y*\$*i*)) *UNIV*
begin

lemma *inner-ge-zero* [*simp*]: $0 \leq \text{real-of } (\text{inner } x \ x)$
by (*auto simp add: inner-vec-def real-sum sum-nonneg*)

lemma *real-scalar-mult2*: $\text{real-of } (\text{inner } x \ x) *_{\mathbb{R}} A = \text{inner } x \ x * A$
by (*auto simp add: inner-vec-def*)
(*metis (mono-tags, lifting) Finite-Cartesian-Product.sum-cong-aux*
real-scalar-mult2 real-sum scaleR-left.sum scale-sum-left)

lemma *i1*: $\text{inner } x \ y = \text{cnj } (\text{inner } y \ x)$
by (*auto simp add: inner-vec-def cnj-sum cnj-mult mult.commute*)
(*meson local.inner-commute*)

lemma *i2*: $\text{inner } (x + y) \ z = \text{inner } x \ z + \text{inner } y \ z$
using *local.inner-left-distrib sum.distrib inner-vec-def* **by force**

lemma *i3*: $\text{inner } (r * s \ x) \ y = r * \text{inner } x \ y$
by (*auto simp add: inner-vec-def scale-sum-right*)

lemma *i4*: **assumes** $\text{inner } x \ x = 0$
shows $x = 0$
proof (*unfold vec-eq-iff, clarify, simp*)
fix *a*
have $0 = \text{real-of } (\sum_{i \in \text{UNIV}} \text{inner-field } (x \ \$ \ i) \ (x \ \$ \ i))$
using *assms* **by** (*simp add: inner-vec-def*)
also have $\dots = (\sum_{i \in \text{UNIV}} \text{real-of } (\text{inner-field } (x \ \$ \ i) \ (x \ \$ \ i)))$
using *real-sum* **by** *auto*
finally have $0 = (\sum_{i \in \text{UNIV}} \text{real-of } (\text{inner-field } (x \ \$ \ i) \ (x \ \$ \ i)))$.
hence $\text{real-of } (\text{inner-field } (x \ \$ \ a) \ (x \ \$ \ a)) = 0$
using *sum-0-all F.inner-ge-zero*
by (*metis (no-types, lifting) finite iso-tuple-UNIV-I*)
then show $x \ \$ \ a = 0$
by (*metis F.inner-eq-zero-iff F.inner-gt-zero-iff real-0*)
qed

lemma *inner-0-0*[*simp*]: $\text{inner } 0 \ 0 = 0$
unfolding *inner-vec-def* **by** *auto*

sublocale *v?*: *inner-product-space* ((**s*) :: 'a \Rightarrow 'a \wedge 'n \Rightarrow 'a \wedge 'n) *inner*
proof (*unfold-locales, auto simp add: real-scalar-mult2*)
fix *x y z*::'a \wedge 'n **and** *r*
show $\text{inner } x \ y = \text{cnj } (\text{inner } y \ x)$ **using** *i1*[*of x y*] **by** *simp*
show $\text{inner } (x + y) \ z = \text{inner } x \ z + \text{inner } y \ z$ **using** *i2* **by** *blast*
show $\text{inner } (r * s \ x) \ y = r * \text{inner } x \ y$ **using** *i3* **by** *blast*
show *i*: $\text{inner } x \ x = 0 \Longrightarrow x = 0$ **using** *i4* **by** *blast*

```

assume  $x \neq 0$ 
thus  $0 < \text{real-of } (\text{inner } x \ x)$  by (metis  $i \langle x \neq 0 \rangle$  inner-0-0 local.inner-ge-zero
local.real-scalar-mult2 mult.commute mult-1-left order.not-eq-order-implies-strict
real-0)
qed
end

```

14.5 Matrices and inner product

```

locale matrix =
  COLS?: vec-real-inner  $\lambda x \ y. x * \text{cnj } y$  inner-cols
  + ROWS?: vec-real-inner  $\lambda x \ y. x * \text{cnj } y$  inner-rows
  for inner-cols ::  $'a^{\wedge}\text{cols}::\{\text{finite}, \text{wellorder}\} \Rightarrow 'a^{\wedge}\text{cols}::\{\text{finite}, \text{wellorder}\} \Rightarrow$ 
 $'a::\{\text{field}, \text{cnj}, \text{real-of-extended}\}$ 
  and inner-rows ::  $'a^{\wedge}\text{rows}::\{\text{finite}, \text{wellorder}\} \Rightarrow 'a^{\wedge}\text{rows}::\{\text{finite}, \text{wellorder}\} \Rightarrow$ 
 $'a$ 
begin

```

lemma *dot-lmul-matrix*: $\text{inner-rows } (x \ v * A) \ y = \text{inner-cols } x \ ((\chi \ i \ j. \text{cnj } (A \ \$ \ i \ \$ \ j)) * v \ y)$

apply (*simp* *add*: *COLS.inner-vec-def* *ROWS.inner-vec-def* *matrix-vector-mult-def*

vector-matrix-mult-def *sum-distrib-right* *cnj-sum* *ac-simps*)

proof (*unfold* *sum-distrib-left*, *subst* *sum.swap*, *rule* *sum.cong*, *simp*)

fix *xa*::*'cols*

show $(\sum_{i \in UNIV}. \text{cnj } (y \ \$ \ i) * (x \ \$ \ xa * A \ \$ \ xa \ \$ \ i))$
 $= (\sum_{n \in UNIV}. x \ \$ \ xa * \text{cnj } (y \ \$ \ n * \text{cnj } (A \ \$ \ xa \ \$ \ n)))$

proof (*rule* *sum.cong*, *simp*)

fix *xb*::*'rows*

show $\text{cnj-class.cnj } (y \ \$ \ xb) * (x \ \$ \ xa * A \ \$ \ xa \ \$ \ xb)$
 $= x \ \$ \ xa * \text{cnj-class.cnj } (y \ \$ \ xb * \text{cnj-class.cnj } (A \ \$ \ xa \ \$ \ xb))$

unfolding *cnj-mult* *cnj-idem* **unfolding** *mult.assoc*

unfolding *mult.commute*[*of* $A \ \$ \ xa \ \$ \ xb$] **by** *auto*

qed

qed

end

14.6 Orthogonal complement generalized

context *inner-product-space*

begin

definition *orthogonal-complement* $W = \{x. \forall y \in W. \text{orthogonal } y \ x\}$

lemma *subspace-orthogonal-complement*: *subspace* (*orthogonal-complement* W)

unfolding *subspace-def* *orthogonal-complement-def*

by (*auto* *simp* *add*: *orthogonal-def* *local.inner-right-distrib*)

lemma *orthogonal-complement-mono*:
assumes *A-in-B*: $A \subseteq B$
shows *orthogonal-complement* $B \subseteq$ *orthogonal-complement* A
proof
fix x **assume** $x: x \in$ *orthogonal-complement* B
show $x \in$ *orthogonal-complement* A **using** x **unfolding** *orthogonal-complement-def*
by (*simp add: orthogonal-def, metis A-in-B in-mono*)
qed

lemma *B-in-orthogonal-complement-of-orthogonal-complement*:
shows $B \subseteq$ *orthogonal-complement* (*orthogonal-complement* B)
by (*auto simp add: orthogonal-complement-def orthogonal-def inner-commute-zero*)

end

14.7 Generalizing projections

context *inner-product-space*
begin

Projection of two vectors: v onto u

definition $proj\ v\ u = scale\ (inner\ v\ u / inner\ u\ u)\ u$

Projection of a onto S

definition $proj\text{-}onto\ a\ S = (sum\ (\lambda x. proj\ a\ x)\ S)$

lemma *vector-sub-project-orthogonal-proj*:
shows $inner\ b\ (x - proj\ x\ b) = 0$
using *vector-sub-project-orthogonal* **unfolding** *proj-def* **by** *simp*

lemma *orthogonal-proj-set*:
assumes $yC: y \in C$ **and** $C: finite\ C$ **and** $p: pairwise\ orthogonal\ C$
shows *orthogonal* ($a - proj\text{-}onto\ a\ C$) y
proof –
have $Cy: C = insert\ y\ (C - \{y\})$ **using** yC
by *blast*
have $fth: finite\ (C - \{y\})$
using C **by** *simp*
show *orthogonal* ($a - proj\text{-}onto\ a\ C$) y
unfolding *orthogonal-def* **unfolding** *proj-onto-def* **unfolding** *proj-def[abs-def]*
unfolding *inner-diff*
unfolding *inner-sum-left*
unfolding *right-minus-eq*
unfolding *sum.remove[OF C yC]*
apply (*clarsimp simp add: inner-commute[of y a]*)
apply (*rule sum.neutral*)
apply *clarsimp*

```

    apply (rule p[unfolded pairwise-def orthogonal-def, rule-format])
    using yC by auto
qed

lemma pairwise-orthogonal-proj-set:
  assumes C: finite C and p: pairwise orthogonal C
  shows pairwise orthogonal (insert (a - proj-onto a C) C)
  by (rule pairwise-orthogonal-insert[OF p], auto simp add: orthogonal-proj-set C
    p)
end

lemma orthogonal-real-eq: RV-inner.orthogonal = real-inner-class.orthogonal
  unfolding RV-inner.orthogonal-def[abs-def]
  unfolding real-inner-class.orthogonal-def[abs-def] ..

```

14.8 Second Part of the Fundamental Theorem of Linear Algebra generalized

```

context matrix
begin

lemma cnj-cnj-matrix[simp]: (χ i j. cnj ((χ i j. cnj (A $ i $ j)) $ i $ j)) = A
  unfolding vec-eq-iff by auto

lemma cnj-transpose[simp]: (χ i j. cnj (transpose A $ i $ j)) = transpose (χ i j.
  cnj (A $ i $ j))
  unfolding vec-eq-iff transpose-def by auto

lemma null-space-orthogonal-complement-row-space:
  fixes A::'a^cols::{finite, wellorder}^rows::{finite, wellorder}
  shows null-space A = COLS.v.orthogonal-complement (row-space (χ i j. cnj (A
    $ i $ j)))
  proof -
    interpret m: matrix inner-rows inner-cols by unfold-locales
    let ?A=(χ i j. cnj (A $ i $ j))
    show ?thesis
    proof (unfold null-space-def COLS.v.orthogonal-complement-def, auto)
      fix x xa assume Ax: A *v x = 0 and xa: xa ∈ row-space (χ i j. cnj (A $ i $
        j))
      obtain y where y: xa = transpose (χ i j. cnj (A $ i $ j)) *v y using xa
      unfolding row-space-eq by blast
      have y2: y v* (χ i j. cnj (A $ i $ j)) = xa
        using transpose-vector y by fastforce
      show COLS.v.orthogonal xa x
        using m.dot-lmul-matrix[of y ?A x]
        unfolding cnj-cnj-matrix Ax ROWS.v.inner-zero-right
        unfolding y2
        unfolding COLS.v.orthogonal-def .
    next

```



```

fix  $x$  assume  $xa$ :  $\forall xa \in \text{row-space } (\chi \ i \ j. \text{cnj } (A \ \$ \ i \ \$ \ j))$ .  $\text{COLS.v.orthogonal } xa$ 
 $x$ 
show  $A * v \ x = 0$ 
using  $xa$  unfolding  $\text{row-space-eq } \text{COLS.v.orthogonal-def}$  using  $\text{COLS.v.inner-eq-zero-iff}$ 
using  $m.\text{dot-lmul-matrix}[of \ - \ ?A]$ 
unfolding  $\text{transpose-vector}$ 
by  $auto$ 
 $(\text{metis } \text{ROWS.i4 } m.\text{cnj-cnj-matrix } m.\text{dot-lmul-matrix } \text{transpose-vector})$ 
qed
qed

```

```

lemma  $\text{left-null-space-orthogonal-complement-col-space}$ :
fixes  $A::'a \wedge \text{cols}::\{\text{finite}, \text{wellorder}\} \wedge \text{rows}::\{\text{finite}, \text{wellorder}\}$ 
shows  $\text{left-null-space } A = \text{ROWS.v.orthogonal-complement } (\text{col-space } (\chi \ i \ j. \text{cnj } (A \ \$ \ i \ \$ \ j)))$ 
proof  $-$ 
interpret  $m$ :  $\text{matrix inner-rows inner-cols}$  by  $\text{unfold-locales}$ 
show  $?thesis$ 
using  $m.\text{null-space-orthogonal-complement-row-space}[of \ \text{transpose } A]$ 
unfolding  $\text{left-null-space-eq-null-space-transpose}$ 
unfolding  $\text{col-space-eq-row-space-transpose}$  by  $auto$ 
qed
end

```

We can get the explicit results for complex and real matrices

```

interpretation  $\text{real-matrix}$ :  $\text{matrix } \lambda x \ y::\text{real} \wedge \text{cols}::\{\text{finite}, \text{wellorder}\}$ .
 $\text{sum } (\lambda i. (x\$i) * (y\$i)) \text{ UNIV } \lambda x \ y. \text{sum } (\lambda i. (x\$i) * (y\$i)) \text{ UNIV}$ 
apply  $(\text{unfold-locales}, \text{auto simp add: } \text{cnj-real-def } \text{real-of-real-def } \text{distrib-right})$ 
using  $\text{not-real-square-gt-zero}$  by  $\text{blast}$ 

```

```

interpretation  $\text{complex-matrix}$ :  $\text{matrix } \lambda x \ y::\text{complex} \wedge \text{cols}::\{\text{finite}, \text{wellorder}\}$ .
 $\text{sum } (\lambda i. (x\$i) * \text{cnj } (y\$i)) \text{ UNIV } \lambda x \ y. \text{sum } (\lambda i. (x\$i) * \text{cnj } (y\$i)) \text{ UNIV}$ 
by  $(\text{unfold-locales}, \text{auto simp add: } \text{distrib-right})$ 

```

```

lemma  $\text{null-space-orthogonal-complement-row-space-complex}$ :
fixes  $A::\text{complex} \wedge \text{cols}::\{\text{finite}, \text{wellorder}\} \wedge \text{rows}::\{\text{finite}, \text{wellorder}\}$ 
shows  $\text{null-space } A = \text{complex-matrix.orthogonal-complement } (\text{row-space } (\chi \ i \ j. \text{cnj } (A \ \$ \ i \ \$ \ j)))$ 
using  $\text{complex-matrix.null-space-orthogonal-complement-row-space}$  .

```

```

lemma  $\text{left-null-space-orthogonal-complement-col-space-complex}$ :
fixes  $A::\text{complex} \wedge \text{cols}::\{\text{finite}, \text{wellorder}\} \wedge \text{rows}::\{\text{finite}, \text{wellorder}\}$ 
shows  $\text{left-null-space } A = \text{complex-matrix.orthogonal-complement } (\text{col-space } (\chi \ i \ j. \text{cnj } (A \ \$ \ i \ \$ \ j)))$ 
using  $\text{complex-matrix.left-null-space-orthogonal-complement-col-space}$  .

```

```

lemma null-space-orthogonal-complement-row-space-reals:
  fixes  $A::\text{real}^{\wedge}\text{cols}::\{\text{finite},\text{wellorder}\}^{\wedge}\text{rows}::\{\text{finite},\text{wellorder}\}$ 
  shows  $\text{null-space } A = \text{real-matrix.orthogonal-complement } (\text{row-space } A)$ 
  using  $\text{real-matrix.null-space-orthogonal-complement-row-space}[of\ A]$ 
  unfolding  $\text{cnj-real-def}$  by ( $\text{simp add: vec-lambda-eta}$ )

```

```

lemma left-null-space-orthogonal-complement-col-space-real:
  fixes  $A::\text{real}^{\wedge}\text{cols}::\{\text{finite},\text{wellorder}\}^{\wedge}\text{rows}::\{\text{finite},\text{wellorder}\}$ 
  shows  $\text{left-null-space } A = \text{real-matrix.orthogonal-complement } (\text{col-space } A)$ 
  using  $\text{real-matrix.left-null-space-orthogonal-complement-col-space}[of\ A]$ 
  by ( $\text{simp add: cnj-real-def vec-lambda-eta}$ )

```

end

15 Improvements to get better performance of the algorithm

```

theory QR-Efficient
imports QR-Decomposition-IArrays
begin

```

15.1 Improvements for computing the Gram Schmidt algorithm and QR decomposition using vecs

Essentially, we try to avoid removing duplicates in each iteration. They will not affect the *sum-list* since the duplicates will be the vector zero.

15.1.1 New definitions

```

definition Gram-Schmidt-column-k-efficient  $A\ k$ 
  = ( $\chi\ a\ b.$  (if  $b = \text{from-nat } k$ 
  then  $\text{column } b\ A - \text{sum-list } (\text{map } (\lambda x. ((\text{column } b\ A \cdot x) / (x \cdot x)) *_R x)$ 
   $((\text{map } (\lambda n. \text{column } (\text{from-nat } n)\ A) [0..<\text{to-nat } b])))$  else  $\text{column } b\ A$ )  $\$ a$ )

```

15.1.2 General properties about *sum-list*

```

lemma sum-list-remdups:
  assumes  $!!i\ j. i < \text{length } xs \wedge j < \text{length } xs \wedge i \neq j$ 
   $\wedge xs ! i = xs ! j \longrightarrow xs ! i = 0 \wedge xs ! j = 0$ 
  shows  $\text{sum-list } (\text{remdups } xs) = \text{sum-list } xs$ 
  using assms
proof (induct  $xs$ )
  case Nil
  thus ?case by auto
next
  case (Cons  $a\ xs$ )
  show ?case
  proof (cases  $a \in \text{set } (xs)$ )

```

```

    case False
      have sum-list (remdups (a # xs)) = sum-list (a # (remdups xs)) by (simp
add: False)
      also have ... = a + sum-list (remdups xs) by auto
      also have ... = a + sum-list xs using Cons.hyps Cons.prem1 False
        by fastforce
      also have ... = sum-list (a # xs) by simp
      finally show ?thesis .
    next
      case True
      have a: a=0 using Cons.hyps Cons.prem1 True
        by (metis Suc-less-eq add.right-neutral add-Suc-right add-gr-0
in-set-conv-nth lessI list.size(4) nat.simp1(3) nth-Cons-0 nth-Cons-Suc)
      have sum-list (remdups (a # xs)) = sum-list (remdups xs) using True by auto
      also have ... = sum-list xs using Cons.hyps Cons.prem1 True
        by fastforce
      also have ... = a + sum-list xs using a by simp
      also have ... = sum-list (a # xs) by simp
      finally show ?thesis .
  qed
qed

```

```

lemma sum-list-remdups-2:
  fixes f:: 'a::{zero, monoid-add}=>'a
  assumes !!i j. i<length xs ^ j<length xs ^ i ≠ j ^ (xs ! i) = (xs ! j)
    → f (xs ! i) = 0 ^ f (xs ! j) = 0
  shows sum-list (map f (remdups xs)) = sum-list (map f xs)
  using assms
proof (induct xs)
  case Nil
  show ?case by auto
next
  case (Cons a xs)
  show ?case
  proof (cases a ∈ set xs)
    case False
    hence sum-list (map f (remdups (a # xs))) = sum-list (map f (a # (remdups
xs)))
      by simp
    also have ... = sum-list (f a # (map f (remdups xs))) by auto
    also have ... = f a + sum-list (map f (remdups xs)) by auto
    also have ... = f a + sum-list (map f xs) using Cons.prem1 Cons.hyps
      using id-apply by fastforce
    also have ... = sum-list (map f (a # xs)) by auto
    finally show ?thesis .
  next
    case True
    have fa-0: f a = 0 using Cons.hyps Cons.prem1 True

```

```

    by (metis Suc-less-eq add.right-neutral add-Suc-right add-gr-0
        in-set-conv-nth lessI list.size(4) nth-Cons-0 nth-Cons-Suc)
  have sum-list (map f (remdups (a # xs))) = sum-list (map f (remdups xs))
    using True by simp
  also have ... = sum-list (map f xs) using Cons.premys Cons.hyps
    using id-apply by fastforce
  also have ... = f a + sum-list (map f xs) using fa-0 by simp
  also have ... = sum-list (map f (a # xs)) by auto
  finally show ?thesis .
qed
qed

```

15.1.3 Proving a code equation to improve the performance

lemma *set-map-column*:

```

  set (map (λn. column (from-nat n) G) [0..to-nat b]) = {column i G | i < b}
proof (auto)
  fix n assume n < to-nat b
  hence from-nat n < b using from-nat-mono to-nat-less-card by fastforce
  thus ∃ i. column (from-nat n) G = column i G ∧ i < b by auto
next
  fix i assume i < b hence ib: to-nat i < to-nat b by (simp add: to-nat-le)
  show column i G ∈ (λn. column (from-nat n) G) ‘ {0..to-nat b}
    unfolding image-def
    by (auto, rule bezI[of - to-nat i], auto simp add: ib)
qed

```

lemma *column-Gram-Schmidt-column-k-repeated-0*:

```

  fixes A::'a::{real-inner} ^'n::{mod-type} ^'m::{mod-type}
  assumes i-not-k: i ≠ k and ik: i < k
  and c-eq: column k (Gram-Schmidt-column-k A (to-nat k))
    = column i (Gram-Schmidt-column-k A (to-nat k))
  and o: pairwise orthogonal {column i A | i < k}
  shows column k (Gram-Schmidt-column-k A (to-nat k)) = 0
  and column i (Gram-Schmidt-column-k A (to-nat k)) = 0
proof -
  have column k (Gram-Schmidt-column-k A (to-nat k))
    = column k A - (∑ x ∈ {column i A | i < k}. (x · column k A / (x · x)) *R x)
    by (rule column-Gram-Schmidt-column-k)
  also have ... = column k A - proj-onto (column k A) {column i A | i < k}
    unfolding proj-onto-def proj-def[abs-def]
    by (metis (no-types, lifting) inner-commute)
  finally have col-k-rw: column k (Gram-Schmidt-column-k A (to-nat k))
    = column k A - proj-onto (column k A) {column i A | i < k} .
  have orthogonal (column k (Gram-Schmidt-column-k A (to-nat k)))
    (column i (Gram-Schmidt-column-k A (to-nat k)))
    unfolding col-k-rw
proof (rule orthogonal-proj-set[OF - - o])
  have column i (Gram-Schmidt-column-k A (to-nat k)) = column i A

```

```

    using column-Gram-Schmidt-column-k'[OF i-not-k] .
    also have ... ∈ {column i A | i. i < k} using assms(2) by blast
    finally show column i (Gram-Schmidt-column-k A (to-nat k)) ∈ {column i A
|i. i < k} .
    show finite {column i A | i. i < k} by auto
qed
thus column k (Gram-Schmidt-column-k A (to-nat k)) = 0
and column i (Gram-Schmidt-column-k A (to-nat k)) = 0
unfolding orthogonal-def c-eq inner-eq-zero-iff by auto
qed

```

```

lemma column-Gram-Schmidt-upt-k-repeated-0':
  fixes A::real^n::{mod-type}^m::{mod-type}
  assumes i-not-k: i≠j and ij: i<j and j: j≤from-nat k
  and c-eq: column j (Gram-Schmidt-upt-k A k)
  = column i (Gram-Schmidt-upt-k A k)
  and k: k<ncols A
  shows column j (Gram-Schmidt-upt-k A k) = 0
  using j c-eq k
proof (induct k)
  case 0
  thus ?case
    using ij least-mod-type
    unfolding from-nat-0
    by (metis (no-types) dual-order.strict-trans1 ij least-mod-type not-less)
next
  case (Suc k)
  have k: k<ncols A using Suc.prem1 unfolding ncols-def by auto
  have i-not-k: i≠from-nat (Suc k) using ij Suc.prem1 by auto
  have col-i-rw: column i (Gram-Schmidt-upt-k A (Suc k)) = column i (Gram-Schmidt-upt-k
A k)
  by (simp add: i-not-k Gram-Schmidt-column-k-def Gram-Schmidt-upt-k-suc col-
umn-def)
  have tn-fn-suc: to-nat (from-nat (Suc k)::'n) = Suc k
  using to-nat-from-nat-id Suc.prem1
  unfolding ncols-def by blast
  show ?case
  proof (cases j=from-nat (Suc k))
  case False
  have jk: j ≤ from-nat k
  by (metis False One-nat-def Suc.prem1 add.right-neutral add-Suc-right
from-nat-suc le-Suc less-le linorder-not-le)
  have col-j-rw: column j (Gram-Schmidt-upt-k A (Suc k)) = column j (Gram-Schmidt-upt-k
A k)
  by (simp add: False Gram-Schmidt-column-k-def Gram-Schmidt-upt-k-suc
column-def)
  have col-j-eq-col-i-k: column j (Gram-Schmidt-upt-k A k) = column i (Gram-Schmidt-upt-k

```

$A\ k$)
using *Suc.prem*s **unfolding** *col-j-rw col-i-rw* **by** *simp*
show *?thesis* **using** *Suc.hyps col-j-eq-col-i-k k jk* **unfolding** *col-j-rw* **by** *blast*
next
case *True*
show *?thesis* **unfolding** *True* **unfolding** *Gram-Schmidt-upt-k-suc*
proof (*rule column-Gram-Schmidt-column-k-repeated-0(1)*)
[*of i from-nat (Suc k) Gram-Schmidt-upt-k A k, unfolded tn-fn-suc*]
have *set-rw*: {*column i (Gram-Schmidt-upt-k A k) | i. i < from-nat (Suc k)*}
= {*column i (Gram-Schmidt-upt-k A k) | i. to-nat i ≤ k*}
by (*metis (mono-tags, opaque-lifting) less-Suc-eq-le less-le not-less tn-fn-suc to-nat-mono*)
show $i \neq \text{from-nat } (Suc\ k)$ **using** *i-not-k* .
show $i < \text{from-nat } (Suc\ k)$ **using** *True ij* **by** *blast*
show *column (from-nat (Suc k)) (Gram-Schmidt-column-k (Gram-Schmidt-upt-k A k) (Suc k)) =*
column i (Gram-Schmidt-column-k (Gram-Schmidt-upt-k A k) (Suc k))
using *Suc.prem*s *True* **by** (*simp add: Gram-Schmidt-upt-k-suc*)
show *pairwise orthogonal {column i (Gram-Schmidt-upt-k A k) | i. i < from-nat (Suc k)}*
(Suc k)
unfolding *set-rw* **by** (*rule orthogonal-Gram-Schmidt-upt-k[OF k]*)
qed
qed
qed

lemma *column-Gram-Schmidt-upt-k-repeated-0*:
fixes $A::\text{real}^n::\{\text{mod-type}\}^m::\{\text{mod-type}\}$
assumes $i\text{-not-}k: i \neq j$ **and** $ij: i < j$ **and** $j \leq k$
and $c\text{-eq}: \text{column } j \text{ (Gram-Schmidt-upt-k } A \text{ (to-nat } k))$
= $\text{column } i \text{ (Gram-Schmidt-upt-k } A \text{ (to-nat } k))$
shows $\text{column } j \text{ (Gram-Schmidt-upt-k } A \text{ (to-nat } k)) = 0$
using *assms column-Gram-Schmidt-upt-k-repeated-0' to-nat-less-card ncols-def*
by (*metis c-eq column-Gram-Schmidt-upt-k-repeated-0'*
from-nat-to-nat-id i-not-k ij j ncols-def to-nat-less-card)

corollary *column-Gram-Schmidt-upt-k-repeated*:
fixes $A::\text{real}^n::\{\text{mod-type}\}^m::\{\text{mod-type}\}$
assumes $i\text{-not-}k: i \neq j$ **and** $ij: i \leq k$ **and** $j \leq k$
and $c\text{-eq}: \text{column } j \text{ (Gram-Schmidt-upt-k } A \text{ (to-nat } k))$
= $\text{column } i \text{ (Gram-Schmidt-upt-k } A \text{ (to-nat } k))$
shows $\text{column } j \text{ (Gram-Schmidt-upt-k } A \text{ (to-nat } k)) = 0$
and $\text{column } i \text{ (Gram-Schmidt-upt-k } A \text{ (to-nat } k)) = 0$
proof –
show $\text{column } j \text{ (Gram-Schmidt-upt-k } A \text{ (to-nat } k)) = 0$
proof (*cases i < j*)

```

    case True
    thus ?thesis using assms column-Gram-Schmidt-upt-k-repeated-0 by metis
next
    case False hence ji: j<i using i-not-k by auto
    thus ?thesis using assms column-Gram-Schmidt-upt-k-repeated-0 by metis
qed
show column i (Gram-Schmidt-upt-k A (to-nat k)) = 0
proof (cases i<j)
  case True
  thus ?thesis using assms column-Gram-Schmidt-upt-k-repeated-0 by metis
next
  case False hence ji: j<i using i-not-k by auto
  thus ?thesis using assms column-Gram-Schmidt-upt-k-repeated-0 by metis
qed
qed

```

lemma *column-Gram-Schmidt-column-k-eq-efficient:*

```

  fixes A::real'n::{mod-type}'m::{mod-type}
  assumes Gram-Schmidt-upt-k A k = foldl Gram-Schmidt-column-k-efficient A
    [0..Suc k]
  and suc-k: Suc k < ncols A
  shows column b (Gram-Schmidt-column-k (Gram-Schmidt-upt-k A k) (Suc k))
    = column b (Gram-Schmidt-column-k-efficient (Gram-Schmidt-upt-k A k) (Suc
    k))
proof (cases b = from-nat (Suc k))
  case False thus ?thesis
    unfolding Gram-Schmidt-column-k-efficient-def Gram-Schmidt-column-k-def
    column-def by auto
next
  case True
  have tn-fn-suc: to-nat (from-nat (Suc k)::'n) = Suc k
    using suc-k to-nat-from-nat-id unfolding ncols-def by auto
  define G where G = Gram-Schmidt-upt-k A k
  let ?f=(λx. (column b G · x / (x · x)) *R x)
  let ?g=(λn. column (from-nat n) G)
  have proj-eq: proj-onto (column b G) {column i G |i. i < b}
    = sum-list (map ?f (map ?g [0..to-nat b]))
  proof -
    have proj-onto (column b G) {column i G |i. i < b} = sum ?f {column i G |i.
    i < b}
    unfolding proj-onto-def proj-def[abs-def] by simp
    also have ... = sum ?f (set (map ?g [0..to-nat b]))
      by (rule sum.cong, auto simp add: set-map-column[symmetric])
    also have ... = sum-list (map ?f (remdups (map ?g [0..to-nat b]))) unfolding
    sum-code ..
    also have ... = sum-list ((map ?f ((map ?g [0..to-nat b])))
  proof (rule sum-list-remdups-2, auto)
    fix i j assume i: i < to-nat b

```

```

and  $j: j < \text{to-nat } b$  and  $ij: i \neq j$ 
and  $\text{col-eq}: \text{column } (\text{from-nat } i) G = \text{column } (\text{from-nat } j) G$ 
and  $\text{col-0}: \text{column } (\text{from-nat } j) G \neq 0$ 
have  $k: \text{to-nat } (\text{from-nat } k)::'n = k$ 
by (metis Suc-lessD ncols-def suc-k to-nat-from-nat-id)
have  $\text{column } (\text{from-nat } j) G = 0$ 
proof (unfold G-def, rule column-Gram-Schmidt-upt-k-repeated(1)
  [of (from-nat i)::'n from-nat j from-nat k A, unfolded k])
have  $\text{from-nat } i < (\text{from-nat } (\text{Suc } k)::'n)$ 
using from-nat-mono[of i Suc k] suc-k i
unfolding True tn-fn-suc ncols-def by simp
thus  $\text{from-nat } i \leq (\text{from-nat } k)::'n$ 
by (metis Suc-lessD True from-nat-mono' i less-Suc-eq-le ncols-def suc-k
tn-fn-suc)
have  $\text{from-nat } j < (\text{from-nat } (\text{Suc } k)::'n)$ 
using from-nat-mono[of j Suc k] suc-k j
unfolding True tn-fn-suc ncols-def by simp
thus  $\text{from-nat } j \leq (\text{from-nat } k)::'n$ 
by (metis Suc-lessD True from-nat-mono' j less-Suc-eq-le ncols-def suc-k
tn-fn-suc)
show  $\text{from-nat } i \neq (\text{from-nat } j)::'n$  using ij i j True suc-k
by (metis (no-types, lifting) dual-order.strict-trans from-nat-eq-imp-eq
ncols-def tn-fn-suc)
show  $\text{column } (\text{from-nat } j) (\text{Gram-Schmidt-upt-k } A k)$ 
  =  $\text{column } (\text{from-nat } i) (\text{Gram-Schmidt-upt-k } A k)$  using G-def col-eq by
auto
qed
thus  $\text{column } b G \cdot \text{column } (\text{from-nat } j) G = 0$  using col-0 by contradiction
qed
finally show ?thesis .
qed
have  $\text{column } b (\text{Gram-Schmidt-column-k } G (\text{Suc } k))$ 
  =  $\text{column } b G - \text{proj-onto } (\text{column } b G) \{ \text{column } i G \mid i. i < b \}$ 
unfolding True
unfolding Gram-Schmidt-column-k-def G-def column-def by vector
also have  $\dots = \text{column } b G$ 
  -  $(\sum x \leftarrow \text{map } (\lambda n. \text{column } (\text{from-nat } n) G) [0..<\text{to-nat } b]. (\text{column } b G \cdot x /$ 
   $(x \cdot x)) *_R x)$ 
unfolding proj-eq ..
also have  $\dots = \text{column } b (\text{Gram-Schmidt-column-k-efficient } G (\text{Suc } k))$ 
unfolding True Gram-Schmidt-column-k-efficient-def G-def column-def by vec-
tor
finally show ?thesis unfolding G-def .
qed

```

lemma *Gram-Schmidt-upt-k-efficient-induction:*

```

fixes  $A::\text{real}^{\sim'n}::\{\text{mod-type}\}^{\sim'm}::\{\text{mod-type}\}$ 
assumes  $\text{Gram-Schmidt-upt-k } A k = \text{foldl } \text{Gram-Schmidt-column-k-efficient } A$ 

```



```

[0..<Suc k]
  and suc-k: Suc k < ncols A
  shows Gram-Schmidt-column-k (Gram-Schmidt-upt-k A k) (Suc k)
  = Gram-Schmidt-column-k-efficient (Gram-Schmidt-upt-k A k) (Suc k)
  using column-Gram-Schmidt-column-k-eq-efficient[OF assms]
  unfolding column-def vec-eq-iff by vector

lemma Gram-Schmidt-upt-k-efficient:
  fixes A::realn::{mod-type}m::{mod-type}
  assumes k: k < ncols A
  shows Gram-Schmidt-upt-k A k = foldl Gram-Schmidt-column-k-efficient A [0..<Suc
k]
  using k
proof (induct k)
  case 0
  have {column i A | i. i < 0} = {} using least-mod-type using leD by auto
  thus ?case
    by (simp, auto simp add: Gram-Schmidt-column-k-efficient-def
      Gram-Schmidt-upt-k-def Gram-Schmidt-column-k-def
      proj-onto-def proj-def vec-eq-iff from-nat-0 to-nat-0)
next
  case (Suc k)
  have Gram-Schmidt-upt-k A (Suc k) = Gram-Schmidt-column-k (Gram-Schmidt-upt-k
A k) (Suc k)
    by (rule Gram-Schmidt-upt-k-suc)
  also have ... = Gram-Schmidt-column-k-efficient (Gram-Schmidt-upt-k A k) (Suc
k)
  proof (rule Gram-Schmidt-upt-k-efficient-induction)
    show Gram-Schmidt-upt-k A k = foldl Gram-Schmidt-column-k-efficient A
[0..<Suc k]
      using Suc.hyps Suc.prem by auto
    show Suc k < ncols A using Suc.prem by auto
  qed
  also have ... = Gram-Schmidt-column-k-efficient
(foldl Gram-Schmidt-column-k-efficient A [0..<Suc k]) (Suc k)
    using Suc.hyps Suc.prem by auto
  also have ... = (foldl Gram-Schmidt-column-k-efficient A [0..<Suc (Suc k)]) by
auto
  finally show ?case .
qed

```

This equation is now more efficient than the original definition of the algorithm, since it is not removing duplicates in each iteration, which is more expensive in time than adding zeros (if there appear duplicates while applying the algorithm, they are zeros and then the *sum-list* is the same in each step).

```

lemma Gram-Schmidt-matrix-efficient[code-unfold]:
  fixes A::realn::{mod-type}m::{mod-type}

```

shows *Gram-Schmidt-matrix* $A = \text{foldl Gram-Schmidt-column-k-efficient } A [0..<\text{ncols } A]$

proof –

have $n: (\text{ncols } A - 1) < \text{ncols } A$ **unfolding** *ncols-def* **by** *auto*
have *Gram-Schmidt-matrix* $A = \text{Gram-Schmidt-upt-k } A (\text{ncols } A - 1)$
unfolding *Gram-Schmidt-matrix-def* ..
also have ... = $\text{foldl Gram-Schmidt-column-k-efficient } A [0..<\text{ncols } A]$
using *Gram-Schmidt-upt-k-efficient*[*OF n*] **unfolding** *ncols-def* **by** *auto*
finally show *?thesis* .

qed

15.2 Improvements for computing the Gram Schmidt algorithm and QR decomposition using immutable arrays

15.2.1 New definitions

definition *Gram-Schmidt-column-k-iarrays-efficient* $A k =$
 $\text{tabulate2 } (nrows\text{-iarray } A) (\text{ncols}\text{-iarray } A) (\lambda a b. \text{let column-b-A} = \text{column-iarray } b \text{ } A \text{ in}$
(if $b = k$ *then* $(\text{column-b-A} - \text{sum-list } (\text{map } (\lambda x. ((\text{column-b-A} \cdot i \ x) / (x \cdot i \ x))$
 $*_R \ x)$
 $((\text{List.map } (\lambda n. \text{column-iarray } n \ A) [0..<b])))$
else $\text{column-b-A}) !! a)$

definition *Gram-Schmidt-matrix-iarrays-efficient* A
 $= \text{foldl Gram-Schmidt-column-k-iarrays-efficient } A [0..<\text{ncols}\text{-iarray } A]$

definition *QR-decomposition-iarrays-efficient* $A =$
(let $Q = \text{divide-by-norm-iarray } (\text{Gram-Schmidt-matrix-iarrays-efficient } A)$
in $(Q, \text{transpose-iarray } Q **i \ A)$

15.2.2 General properties

lemma *tabulate2-nth*:

assumes $i: i < nr$ **and** $j: j < nc$
shows $(\text{tabulate2 } nr \ nc \ f) !! i !! j = f \ i \ j$
unfolding *tabulate2-def* **using** *i j nth-map* **by** *auto*

lemma *vec-to-iarray-minus*[*code-unfold*]:

$\text{vec-to-iarray } (a - b) = (\text{vec-to-iarray } a) - (\text{vec-to-iarray } b)$
unfolding *vec-to-iarray-def*
unfolding *minus-iarray-def* *Let-def* **by** *auto*

lemma *vec-to-iarray-minus-nth*:

assumes $A: i < IArray.length (\text{vec-to-iarray } A)$
and $B: i < IArray.length (\text{vec-to-iarray } B)$
shows $(\text{vec-to-iarray } A - \text{vec-to-iarray } B) !! i$
 $= \text{vec-to-iarray } A !! i - \text{vec-to-iarray } B !! i$

proof –

```

have  $i$ :  $i < \text{CARD}'b$  using  $A$  unfolding  $\text{vec-to-iarray-def}$  by  $\text{auto}$ 
have  $i2$ :  $i < \text{CARD}'c$  using  $B$  unfolding  $\text{vec-to-iarray-def}$  by  $\text{auto}$ 
have  $i\text{-length}$ :  $i < \text{length } [0..<\max \text{CARD}'b \text{ CARD}'c]$  using  $i \ i2$  by  $\text{auto}$ 
have  $i\text{-nth}$ :  $[0..<\max \text{CARD}'b \text{ CARD}'c] ! i = i$  using  $i\text{-length}$  by  $\text{auto}$ 
let  $?f = (\lambda a. \text{map } (\lambda a. \text{if } a < \text{CARD}'b \text{ then } \text{IArray}$ 
   $(\text{map } (\lambda i. A \ \$ \ \text{from-nat } i) [0..<\text{CARD}'b]) !! a \ \text{else } 0) [0..<\max \text{CARD}'b$ 
 $\text{CARD}'c]) !$ 
   $a - \text{map } (\lambda a. \text{if } a < \text{CARD}'c \text{ then}$ 
     $\text{IArray } (\text{map } (\lambda i. B \ \$ \ \text{from-nat } i) [0..<\text{CARD}'c]) !! a \ \text{else } 0) [0..<\max$ 
 $\text{CARD}'b \ \text{CARD}'c] ! a)$ 
  have  $(\text{vec-to-iarray } A - \text{vec-to-iarray } B) = (\text{IArray } (\text{map } (\lambda i. A \ \$ \ \text{from-nat } i)$ 
 $[0..<\text{CARD}'b]))$ 
     $- \text{IArray } (\text{map } (\lambda i. B \ \$ \ \text{from-nat } i) [0..<\text{CARD}'c]))$ 
    unfolding  $\text{vec-to-iarray-def}$  by  $\text{auto}$ 
  also have  $\dots = \text{IArray } (\text{map } ?f [0..<\max \text{CARD}'b \ \text{CARD}'c])$ 
    unfolding  $\text{minus-iarray-def}$   $\text{Let-def}$  by  $\text{simp}$ 
  also have  $\dots !! i = A \ \$ \ \text{from-nat } i - B \ \$ \ \text{from-nat } i$ 
    using  $i\text{-length}$  using  $\text{nth-map } i \ i2$  by  $\text{auto}$ 
  also have  $\dots = \text{vec-to-iarray } A !! i - \text{vec-to-iarray } B !! i$ 
    by  $(\text{metis } i \ i2 \ \text{vec-to-iarray-nth})$ 
  finally show  $?thesis$  .
qed

```

lemma $\text{sum-list-map-vec-to-iarray}$:

```

assumes  $xs \neq []$ 
shows  $\text{sum-list } (\text{map } (\text{vec-to-iarray} \circ f) \ xs) = \text{vec-to-iarray } (\text{sum-list } (\text{map } f \ xs))$ 
using  $\text{assms}$ 
proof  $(\text{induct } xs)$ 
  case  $\text{Nil}$ 
    thus  $?case$  unfolding  $\text{o-def}$  by  $\text{auto}$ 
next
  case  $(\text{Cons } a \ xs)$ 
    show  $?case$ 
    proof  $(\text{cases } xs = [])$ 
      case  $\text{True}$ 
        have  $l\text{-rw}$ :  $\text{sum-list } (\text{map } (\text{vec-to-iarray} \circ f) \ xs) = \text{IArray} []$ 
          unfolding  $\text{True}$  by  $(\text{simp } \text{add: } \text{zero-iarray-def})$ 
        have  $\text{sum-list } (\text{map } (\text{vec-to-iarray} \circ f) \ (a \ \# \ xs))$ 
           $= \text{sum-list } ((\text{vec-to-iarray} \circ f) \ a \ \# \ \text{map } (\text{vec-to-iarray} \circ f) \ xs)$ 
          by  $\text{simp}$ 
        also have  $\dots = (\text{vec-to-iarray} \circ f) \ a + \text{sum-list } (\text{map } (\text{vec-to-iarray} \circ f) \ xs)$ 
by  $\text{simp}$ 
        also have  $\dots = \text{vec-to-iarray } (f \ a) + \text{IArray} []$  unfolding  $l\text{-rw}$  by  $\text{auto}$ 
        also have  $\dots = \text{vec-to-iarray } (f \ a) + \text{vec-to-iarray } (0::('b,'c) \ \text{vec})$ 
          unfolding  $\text{plus-iarray-def}$   $\text{Let-def}$   $\text{vec-to-iarray-def}$  by  $\text{auto}$ 
        also have  $\dots = \text{vec-to-iarray } (\text{sum-list } (\text{map } f) \ (a \ \# \ xs))$ 
          unfolding  $\text{True}$  unfolding  $\text{plus-iarray-def}$   $\text{Let-def}$   $\text{vec-to-iarray-def}$  by  $\text{auto}$ 
        finally show  $?thesis$  .

```

```

next
  case False
  have sum-list (map (vec-to-iarray  $\circ$  f) (a # xs))
    = sum-list ((vec-to-iarray  $\circ$  f) a # map (vec-to-iarray  $\circ$  f) xs)
    by simp
  also have ... = (vec-to-iarray  $\circ$  f) a + sum-list (map (vec-to-iarray  $\circ$  f) xs)
by simp
  also have ... = (vec-to-iarray  $\circ$  f) a + vec-to-iarray (sum-list (map f xs))
    using Cons.prems Cons.hyps False by presburger
  also have ... = vec-to-iarray (f a) + vec-to-iarray (sum-list (map f xs)) by auto
  also have ... = vec-to-iarray (f a + (sum-list (map f xs)))
    by (simp add: vec-to-iarray-plus)
  also have ... = vec-to-iarray (sum-list (map (f) (a # xs))) by simp
  finally show ?thesis .
qed
qed

```

15.2.3 Proving the equivalence

lemma *matrix-to-iarray-Gram-Schmidt-column-k-efficient*:

```

  fixes A::realn::{mod-type}m::{mod-type}
  assumes k: k < ncols A
  shows matrix-to-iarray (Gram-Schmidt-column-k-efficient A k)
    = Gram-Schmidt-column-k-iarrays-efficient (matrix-to-iarray A) k
proof (unfold iarray-exhaust2 list-eq-iff-nth-eq, rule conjI, auto,
  unfold IArray.sub-def[symmetric] IArray.length-def[symmetric])
  show IArray.length (matrix-to-iarray (Gram-Schmidt-column-k-efficient A k))
    = IArray.length (Gram-Schmidt-column-k-iarrays-efficient (matrix-to-iarray A)
  k)
    unfolding matrix-to-iarray-def Gram-Schmidt-column-k-iarrays-efficient-def
tabulate2-def
    unfolding nrows-iarray-def by auto
  fix i
  show i < IArray.length (matrix-to-iarray (Gram-Schmidt-column-k-efficient A
  k))  $\implies$ 
    IArray.length (matrix-to-iarray (Gram-Schmidt-column-k-efficient A k) !! i) =
    IArray.length (Gram-Schmidt-column-k-iarrays-efficient (matrix-to-iarray A) k
  !! i)
    by (simp add: matrix-to-iarray-def Gram-Schmidt-column-k-iarrays-efficient-def
    Gram-Schmidt-column-k-efficient-def tabulate2-def ncols-iarray-def
    nrows-iarray-def vec-to-iarray-def)
  fix i ia
  assume i: i < IArray.length (matrix-to-iarray (Gram-Schmidt-column-k-efficient
  A k))
    and ia: ia < IArray.length (matrix-to-iarray (Gram-Schmidt-column-k-efficient
  A k) !! i)
  show matrix-to-iarray (Gram-Schmidt-column-k-efficient A k) !! i !! ia
    = Gram-Schmidt-column-k-iarrays-efficient (matrix-to-iarray A) k !! i !! ia

```

```

proof –
  let ?f=(λa b. let column-b-A = column-iarray b (matrix-to-iarray A)
    in (if b = k then column-b-A
      – (∑ x←map (λn. column-iarray n (matrix-to-iarray A)) [0..b].
        (column-b-A · i x / (x · i x)) *R x) else column-b-A !! a)
  have i2: i < nrows A using i unfolding nrows-def matrix-to-iarray-def by auto
  have ia2: ia < ncols A
  using ia unfolding ncols-def matrix-to-iarray-def o-def vec-to-iarray-def
  by (metis i2 ia length-vec-to-iarray nrows-def to-nat-from-nat-id vec-matrix)
  have Gram-Schmidt-column-k-iarrays-efficient (matrix-to-iarray A) k !! i !! ia
= ?f i ia
  unfolding Gram-Schmidt-column-k-iarrays-efficient-def
proof (rule tabulate2-nth)
  show i < nrows-iarray (matrix-to-iarray A)
  using i2 unfolding matrix-to-iarray-nrows .
  show ia < ncols-iarray (matrix-to-iarray A)
  using ia2 unfolding matrix-to-iarray-ncols .
qed
also have ... = (Gram-Schmidt-column-k-efficient A) $ (from-nat i) $ (from-nat
ia)
  unfolding Gram-Schmidt-column-k-efficient-def Let-def
proof (auto)
  assume ia-neq-k: ia ≠ k and f-eq: (from-nat ia::'n) = from-nat k
  have ia = k using f-eq by (metis assms from-nat-eq-imp-eq ia2 ncols-def)
  thus IArray.list-of (column-iarray ia (matrix-to-iarray A)) ! i =
  column (from-nat k) A $ from-nat i –
  sum-list (map ((λx. (column (from-nat k) A · x / (x · x)) *R x)
  ○ (λn. column (from-nat n) A)) [0..to-nat (from-nat k)]) $ from-nat i
  using ia-neq-k by contradiction
next
  assume ia ≠ k
  thus IArray.list-of (column-iarray ia (matrix-to-iarray A)) ! i
  = column (from-nat ia) A $ from-nat i
  by (metis IArray.sub-def i ia2 length-eq-card-rows to-nat-from-nat-id
vec-to-iarray-column' vec-to-iarray-nth')
next
  assume ia = k
  let ?f=λx. ((column (from-nat k) A) · (column (from-nat x) A)) /
  ((column (from-nat x) A) · (column (from-nat x) A)) *R (column (from-nat
x) A)
  let ?l=sum-list (map ?f [0..k])
  show IArray.list-of
  (column-iarray k (matrix-to-iarray A) –
  sum-list (map ((λx. (column-iarray k (matrix-to-iarray A) · i x / (x · i x))
  *R x)
  ○ (λn. column-iarray n (matrix-to-iarray A)) [0..k])) ! i =
  column (from-nat k) A $ from-nat i –
  sum-list (map ((λx. (column (from-nat k) A · x / (x · x)) *R x)
  ○ (λn. column (from-nat n) A)) [0..to-nat (from-nat k)::'n]) $ from-nat i

```

```

proof (cases k=0)
  case True
  show ?thesis
    unfolding vec-to-iarray-column'[OF k, symmetric]
    unfolding True from-nat-0 to-nat-0
  by (auto, metis IArray.sub-def i2 minus-zero-iarray nrows-def vec-to-iarray-nth)

next
  case False
  have tn-fn-k: to-nat (from-nat k::'n) = k
    by (metis assms from-nat-to-nat ncols-def)
  have column-rw: column-iarray k (matrix-to-iarray A)
    = vec-to-iarray (column (from-nat k) A)
    by (rule vec-to-iarray-column'[symmetric, OF k])
  have sum-list-rw: (∑ x←[0..R column-iarray x (matrix-to-iarray
A))
    = vec-to-iarray ?l
  proof -
    have (∑ x←[0..R column-iarray x (matrix-to-iarray
A))
    = sum-list (map (vec-to-iarray ∘ ?f) [0..proof (unfold interv-sum-list-conv-sum-set-nat, rule sum.cong, auto)
    fix x assume x < k
    hence x: x < ncols A using k by auto
  show (column-iarray k (matrix-to-iarray A) ·i column-iarray x (matrix-to-iarray
A) /
A) /
    (column-iarray x (matrix-to-iarray A) ·i column-iarray x (matrix-to-iarray
A)) *R
    column-iarray x (matrix-to-iarray A) =
    vec-to-iarray ((column (from-nat k) A · column (from-nat x) A /
    (column (from-nat x) A · column (from-nat x) A)) *R column (from-nat
x) A)
    unfolding vec-to-iarray-scaleR vec-to-iarray-inner
  unfolding column-rw unfolding vec-to-iarray-column'[OF x, symmetric]
  ..
  qed
  also have ... = vec-to-iarray (sum-list (map ?f [0..by (rule sum-list-map-vec-to-iarray, auto simp add: False)
  finally show ?thesis .
qed

have IArray.list-of

```

```

      (column-iarray k (matrix-to-iarray A) -
sum-list (map ((λx. (column-iarray k (matrix-to-iarray A) ·i x / (x ·i x))
*_R x)
  ○ (λn. column-iarray n (matrix-to-iarray A))) [0..<k])) ! i =
      (column-iarray k (matrix-to-iarray A) -
      (∑ x←[0..<k]. (column-iarray k (matrix-to-iarray A) ·i column-iarray x
(matrix-to-iarray A) /
      (column-iarray x (matrix-to-iarray A) ·i column-iarray x (matrix-to-iarray
A)))) *_R
      column-iarray x (matrix-to-iarray A))) !! i
      unfolding vec-to-iarray-inner tn-fn-k o-def
      unfolding IArray.sub-def[symmetric] ..
      also have ... = (vec-to-iarray (column (from-nat k) A) - vec-to-iarray ?l)
!! i
      unfolding sum-list-rw unfolding column-rw ..
      also have ... = ((vec-to-iarray (column (from-nat k) A)) !! i) - (vec-to-iarray
?l !! i)
      proof (rule vec-to-iarray-minus-nth)
      show i < IArray.length (vec-to-iarray (column (from-nat k) A))
      by (metis i2 length-vec-to-iarray nrows-def)
      show i < IArray.length (vec-to-iarray ?l)
      by (metis (no-types, lifting) i2 length-vec-to-iarray nrows-def)
      qed
      also have ... = column (from-nat k) A $ from-nat i - ?l $ from-nat i
      unfolding column-rw
      by (metis (no-types, lifting) i2 nrows-def vec-to-iarray-nth)
      also have ... = column (from-nat k) A $ from-nat i -
      sum-list (map ((λx. (column (from-nat k) A · x / (x · x)) *_R x)
  ○ (λn. column (from-nat n) A)) [0..<to-nat (from-nat k::'n)]) $ from-nat i
      unfolding o-def tn-fn-k ..
      finally show IArray.list-of
      (column-iarray k (matrix-to-iarray A) -
sum-list (map ((λx. (column-iarray k (matrix-to-iarray A) ·i x / (x ·i x))
*_R x)
  ○ (λn. column-iarray n (matrix-to-iarray A))) [0..<k])) ! i =
      column (from-nat k) A $ from-nat i -
      sum-list (map ((λx. (column (from-nat k) A · x / (x · x)) *_R x)
  ○ (λn. column (from-nat n) A)) [0..<to-nat (from-nat k::'n)]) $ from-nat
i .
      qed
      qed
      also have ... = matrix-to-iarray (Gram-Schmidt-column-k-efficient A k) !! i !!
ia
      using matrix-to-iarray-nth[of (Gram-Schmidt-column-k-efficient A k) from-nat
i from-nat ia]
      using ia2 i2
      unfolding to-nat-from-nat-id[OF i2[unfolded nrows-def]]
      unfolding to-nat-from-nat-id[OF ia2[unfolded ncols-def]] by simp
      finally show ?thesis ..

```

qed
qed

lemma *matrix-to-iarray-Gram-Schmidt-upt-k-efficient*:

fixes $A::\text{real}^n::\{\text{mod-type}\}^m::\{\text{mod-type}\}$

assumes $k: k < \text{ncols } A$

shows *matrix-to-iarray* (*Gram-Schmidt-upt-k* A k)

= *foldl Gram-Schmidt-column-k-iarrays-efficient* (*matrix-to-iarray* A) [$0..< \text{Suc } k$]

using *assms*

proof (*induct k*)

case 0

have *zero-le*: $0 < \text{ncols } A$ **unfolding** *ncols-def* **by** *simp*

thus $?case$ **unfolding** *Gram-Schmidt-upt-k-efficient*[*OF zero-le*] **unfolding** *Gram-Schmidt-upt-k-efficient*
by (*simp add: matrix-to-iarray-Gram-Schmidt-column-k-efficient*[*OF 0.prem*])

next

case (*Suc k*)

let $?G = \text{foldl Gram-Schmidt-column-k-iarrays-efficient}$ (*matrix-to-iarray* A)

have $k: k < \text{ncols}$ (*Gram-Schmidt-upt-k* A k) **using** *Suc.prem* **unfolding** *ncols-def*
by *simp*

have $k2: \text{Suc } k < \text{ncols}$ (*Gram-Schmidt-upt-k* A k) **using** *Suc.prem* **unfolding**
ncols-def .

have *list-rw*: $[0..< \text{Suc} (\text{Suc } k)] = [0..< \text{Suc } k] @ [(\text{Suc } k)]$ **by** *simp*

have *hyp*: *matrix-to-iarray* (*Gram-Schmidt-upt-k* A k) = $?G$ [$0..< \text{Suc } k$]

by (*metis Suc.hyps Suc.prem Suc-lessD*)

show *matrix-to-iarray* (*Gram-Schmidt-upt-k* A (*Suc k*)) = $?G$ [$0..< \text{Suc} (\text{Suc } k)$]

unfolding *Gram-Schmidt-upt-k-def*

unfolding *list-rw*

unfolding *foldl-append*

unfolding *foldl.simps*

unfolding *Gram-Schmidt-upt-k-def*[*symmetric*]

unfolding *hyp*[*symmetric*]

using *matrix-to-iarray-Gram-Schmidt-column-k-efficient*

by (*metis* (*no-types*) *Gram-Schmidt-upt-k-efficient* *Gram-Schmidt-upt-k-efficient-induction*

Suc.prem k matrix-to-iarray-Gram-Schmidt-column-k-efficient ncols-def)

qed

lemma *matrix-to-iarray-Gram-Schmidt-matrix-efficient*[*code-unfold*]:

fixes $A::\text{real}^n::\{\text{mod-type}\}^m::\{\text{mod-type}\}$

shows *matrix-to-iarray* (*Gram-Schmidt-matrix* A)

= *Gram-Schmidt-matrix-iarrays-efficient* (*matrix-to-iarray* A)

proof –

have $n: \text{ncols } A - 1 < \text{ncols } A$ **unfolding** *ncols-def* **by** *auto*

thus $?thesis$

unfolding *Gram-Schmidt-matrix-iarrays-efficient-def* *Gram-Schmidt-matrix-def*


```

using matrix-to-iarray-Gram-Schmidt-upt-k-efficient[OF n]
unfolding matrix-to-iarray-ncols by auto
qed

```

```

lemma QR-decomposition-iarrays-efficient[code]:
  QR-decomposition-iarrays (matrix-to-iarray A)
  = QR-decomposition-iarrays-efficient (matrix-to-iarray A)
unfolding QR-decomposition-iarrays-def QR-decomposition-iarrays-efficient-def
Let-def
unfolding matrix-to-iarray-Gram-Schmidt-matrix-efficient[symmetric]
unfolding matrix-to-iarray-Gram-Schmidt-matrix ..

```

15.3 Other code equations that improve the performance

```

lemma inner-iarray-code[code]:
  inner-iarray A B = sum-list (map ( $\lambda n. A !! n * B !! n$ ) [0..IArray.length A])
proof –
  have set-Eq: {0..IArray.length A} = set ([0..IArray.length A]) using atLeast-LessThan-upt by blast
  have inner-iarray A B = sum ( $\lambda n. A !! n * B !! n$ ) {0..IArray.length A}
  unfolding inner-iarray-def ..
  also have ... = sum ( $\lambda n. A !! n * B !! n$ ) (set [0..IArray.length A])
  unfolding set-Eq ..
  also have ... = sum-list (map ( $\lambda n. A !! n * B !! n$ ) [0..IArray.length A])
  unfolding sum-set-upt-conv-sum-list-nat ..
  finally show ?thesis .
qed

```

```

definition Gram-Schmidt-column-k-iarrays-efficient2 A k =
  tabulate2 (nrows-iarray A) (ncols-iarray A)
  (let col-k = column-iarray k A;
    col = (col-k – sum-list (map ( $\lambda x. ((col-k \cdot i x) / (x \cdot i x)) *_{\mathbb{R}} x$ )
      ((List.map ( $\lambda n. column-iarray\ n\ A$ ) [0..k])))
    in ( $\lambda a\ b. (if\ b = k\ then\ col\ else\ column-iarray\ b\ A) !! a$ ))

```

```

lemma Gram-Schmidt-column-k-iarrays-efficient-eq[code]: Gram-Schmidt-column-k-iarrays-efficient
A k
  = Gram-Schmidt-column-k-iarrays-efficient2 A k
unfolding Gram-Schmidt-column-k-iarrays-efficient-def
unfolding Gram-Schmidt-column-k-iarrays-efficient2-def
unfolding Let-def tabulate2-def
by simp

```

end