

QR Decomposition

By Jose Divasón and Jesús Aransay*

April 10, 2026

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	5
2.1	Definitions of vector projection and projection of a vector onto a set.	5
2.2	Properties	5
2.3	Orthogonal Complement	6
2.4	Normalization of vectors	7
3	The Gram-Schmidt algorithm	7
3.1	Gram-Schmidt algorithm	7
3.1.1	First way	8
3.1.2	Second way	9
3.1.3	Third way	9
3.1.4	Examples of execution	14

*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	15
4.1	The QR Decomposition of a matrix	15
4.1.1	Divide a vector by its norm	16
4.1.2	The QR Decomposition	17
5	Least Squares Approximation	21
5.1	Second part of the Fundamental Theorem of Linear Algebra .	21
5.2	Least Squares Approximation	21
6	Examples of execution using floats	24
6.0.1	Examples	24
7	Examples of execution using symbolic computation	25
7.1	Execution of the QR decomposition using symbolic computation	25
7.1.1	Some previous definitions and lemmas	25
7.1.2	Examples	25
8	IArray Addenda QR	27
8.1	Some previous instances	28
8.2	Some previous definitions and properties for IArrays	29
8.2.1	Lemmas	29
8.2.2	Definitions	29
8.3	Code generation	29
9	Matrices as nested IArrays	29
9.1	Isomorphism between matrices implemented by vecs and matrices implemented by iarrays	29
9.1.1	Isomorphism between vec and iarray	29
9.1.2	Isomorphism between matrix and nested iarrays	30
9.2	Definition of operations over matrices implemented by iarrays	31
9.2.1	Properties of previous definitions	32
9.3	Definition of elementary operations	34
9.3.1	Code generator	34
10	Gram Schmidt over IArrays	36
10.1	Some previous definitions, lemmas and instantiations about iarrays	36
10.2	Inner mult over real iarrays	37
10.3	Gram Schmidt over IArrays	38
11	QR Decomposition over iarrays	39
11.1	QR Decomposition refinement over iarrays	39
12	Examples of execution using floats and IArrays	40
12.1	Examples	40

13 Examples of execution using symbolic computation and iarrays	41
13.1 Execution of the QR decomposition using symbolic computation and iarrays	41
13.1.1 Examples	42
14 Generalization of the Second Part of the Fundamental Theorem of Linear Algebra	45
14.1 Conjugate class	45
14.2 Real_of_extended class	46
14.3 Generalizing HMA	47
14.3.1 Inner product spaces	47
14.3.2 Orthogonality	49
14.4 Vecs as inner product spaces	50
14.5 Matrices and inner product	51
14.6 Orthogonal complement generalized	51
14.7 Generalizing projections	51
14.8 Second Part of the Fundamental Theorem of Linear Algebra generalized	52
15 Improvements to get better performance of the algorithm	53
15.1 Improvements for computing the Gram Schmidt algorithm and QR decomposition using vecs	53
15.1.1 New definitions	54
15.1.2 General properties about <i>sum-list</i>	54
15.1.3 Proving a code equation to improve the performance	54
15.2 Improvements for computing the Gram Schmidt algorithm and QR decomposition using immutable arrays	56
15.2.1 New definitions	56
15.2.2 General properties	56
15.2.3 Proving the equivalence	57
15.3 Other code equations that improve the performance	57

1 Miscellaneous file for the QR algorithm

```

theory Miscellaneous-QR
imports
  Gauss-Jordan.Determinants2
  Gauss-Jordan.Inverse
begin

```

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

```

lemma [code abstract]:
   $vec\_nth (c *_R x) = (\lambda i. c *_R (x\$i)) \langle proof \rangle$ 

```

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 : *to-nat* $i < k$
and k : $k < \text{CARD}('a)$
shows $i < \text{from-nat } k$
 $\langle \text{proof} \rangle$

Some properties about orthogonal matrices.

lemma *orthogonal-mult*:
assumes *orthogonal* a b
shows *orthogonal* $(x *_R a)$ $(y *_R b)$
 $\langle \text{proof} \rangle$

lemma *orthogonal-matrix-is-orthogonal*:
fixes $A::\text{real}^n{}^n$
assumes o : *orthogonal-matrix* A
shows $(\text{pairwise orthogonal (columns } A))$
 $\langle \text{proof} \rangle$

lemma *orthogonal-matrix-norm*:
fixes $A::\text{real}^n{}^n$
assumes o : *orthogonal-matrix* A
shows $\text{norm (column } i \text{ } A) = 1$
 $\langle \text{proof} \rangle$

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

lemma *orthogonal-matrix-intro*:
fixes $A::\text{real}^n{}^n$
assumes p : $(\text{pairwise orthogonal (columns } A))$
and n : $\forall i. \text{norm (column } i \text{ } A) = 1$
and c : $\text{card (columns } A) = \text{ncols } A$
shows *orthogonal-matrix* A
 $\langle \text{proof} \rangle$

lemma *orthogonal-matrix2*:
fixes $A::\text{real}^n{}^n$
shows *orthogonal-matrix* $A = ((\text{pairwise orthogonal (columns } A)) \wedge (\forall i. \text{norm (column } i \text{ } A) = 1) \wedge (\text{card (columns } A) = \text{ncols } A))$
 $\langle \text{proof} \rangle$

lemma *orthogonal-matrix'*: *orthogonal-matrix* ($Q :: \text{real } ^n ^n$) \longleftrightarrow $Q ** \text{transpose } Q = \text{mat } 1$
 ⟨*proof*⟩

lemma *orthogonal-matrix-intro2*:
fixes $A :: \text{real } ^n ^n$
assumes p : (*pairwise orthogonal* (*rows* A))
and n : $\forall i. \text{norm } (\text{row } i \ A) = 1$
and c : $\text{card } (\text{rows } A) = \text{nrows } A$
shows *orthogonal-matrix* A
 ⟨*proof*⟩

lemma *is-basis-imp-full-rank*:
fixes $A :: 'a :: \{\text{field}\} ^{\text{cols}} :: \{\text{mod-type}\} ^{\text{rows}} :: \{\text{mod-type}\}$
assumes b : *is-basis* (*columns* A)
and c : $\text{card } (\text{columns } A) = \text{ncols } A$
shows $\text{rank } A = \text{ncols } A$
 ⟨*proof*⟩

lemma *card-columns-le-ncols*:
 $\text{card } (\text{columns } A) \leq \text{ncols } A$
 ⟨*proof*⟩

lemma *full-rank-imp-is-basis*:
fixes $A :: 'a :: \{\text{field}\} ^n :: \{\text{mod-type}\} ^n :: \{\text{mod-type}\}$
assumes r : $\text{rank } A = \text{ncols } A$
shows $\text{is-basis } (\text{columns } A) \wedge \text{card } (\text{columns } A) = \text{ncols } A$
 ⟨*proof*⟩

lemma *full-rank-imp-is-basis2*:
fixes $A :: 'a :: \{\text{field}\} ^n :: \{\text{mod-type}\} ^m :: \{\text{mod-type}\}$
assumes r : $\text{rank } A = \text{ncols } A$
shows $\text{vec.independent } (\text{columns } A) \wedge \text{vec.span } (\text{columns } A) = \text{col-space } A$
 $\wedge \text{card } (\text{columns } A) = \text{ncols } A$
 ⟨*proof*⟩

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

lemma *full-col-rank-imp-independent-columns*:
fixes $A :: 'a :: \{\text{field}\} ^n :: \{\text{mod-type}\} ^m :: \{\text{mod-type}\}$
assumes $\text{rank } A = \text{ncols } A$
shows $\text{vec.independent } (\text{columns } A)$
 ⟨*proof*⟩

```

lemma matrix-vector-right-distrib-minus:
  fixes  $A::'a::\{\text{ring-1}\}^{\wedge n \wedge m}$ 
  shows  $A * v (b - c) = (A * v b) - (A * v c)$ 
   $\langle \text{proof} \rangle$ 

lemma inv-matrix-vector-mul-left:
  assumes  $i: \text{invertible } A$ 
  shows  $(A * v x = A * v y) = (x=y)$ 
   $\langle \text{proof} \rangle$ 

lemma norm-mult-vec:
  fixes  $a::(\text{real}, 'b::\text{finite}) \text{ vec}$ 
  shows  $\text{norm } (x \cdot x) = \text{norm } x * \text{norm } x$ 
   $\langle \text{proof} \rangle$ 

lemma norm-equivalence:
  fixes  $A::\text{real}^{\wedge n \wedge m}$ 
  shows  $((\text{transpose } A) * v (A * v x) = 0) \longleftrightarrow (A * v x = 0)$ 
   $\langle \text{proof} \rangle$ 

lemma invertible-transpose-mult:
  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  $\text{invertible } (\text{transpose } A ** A)$ 
   $\langle \text{proof} \rangle$ 

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  $\text{invertible } A$  and  $\text{invertible } B$ 
  shows  $\text{matrix-inv } (A ** B) = \text{matrix-inv } B ** \text{matrix-inv } A$ 
   $\langle \text{proof} \rangle$ 

lemma invertible-transpose:
  fixes  $A::'a::\{\text{field}\}^{\wedge n \wedge n}$ 
  assumes  $\text{invertible } A$ 
  shows  $\text{invertible } (\text{transpose } A)$ 
   $\langle \text{proof} \rangle$ 

```

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:  $(\text{span } S = \text{span } T) = (S \subseteq \text{span } T \wedge T \subseteq \text{span } S)$ 
   $\langle \text{proof} \rangle$ 

```

end

lemma *basis-orthogonal*:

fixes $B :: 'a::\text{real-inner set}$

assumes $fB: \text{finite } B$

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 $\exists C. ?P B C)$

$\langle \text{proof} \rangle$

lemma *op-vec-scaleR*: $(*s) = (*_R)$

$\langle \text{proof} \rangle$

end

2 Projections

theory *Projections*

imports

Miscellaneous-QR

begin

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

definition $\text{proj } v \ u = (v \cdot u / (u \cdot u)) *_R \ u$

definition $\text{proj-onto } a \ S = (\text{sum } (\lambda x. \text{proj } a \ x) \ S)$

2.2 Properties

lemma *proj-onto-sum-rw*:

$\text{sum } (\lambda x. (x \cdot v / (x \cdot x)) *_R \ x) \ A = \text{sum } (\lambda x. (v \cdot x / (x \cdot x)) *_R \ x) \ A$
 $\langle \text{proof} \rangle$

lemma *vector-sub-project-orthogonal-proj*:

fixes $b \ x :: 'a::\text{euclidean-space}$

shows $\text{inner } b \ (x - \text{proj } x \ b) = 0$

$\langle \text{proof} \rangle$

lemma *orthogonal-proj-set*:

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

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

$\langle \text{proof} \rangle$

lemma *pairwise-orthogonal-proj-set*:

assumes $C: \text{finite } C$ **and** $p: \text{pairwise orthogonal } C$

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

<proof>

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* $x \ y$

<proof>

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

<proof>

lemma *orthogonal-complement-mono*:

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

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

<proof>

lemma *B-in-orthogonal-complement-of-orthogonal-complement*:

shows $B \subseteq \text{orthogonal-complement} (\text{orthogonal-complement } B)$

<proof>

lemma *pythagorean-theorem-norm*:

assumes *o*: *orthogonal* $x \ y$

shows $\text{norm } (x+y)^2 = \text{norm } x^2 + \text{norm } y^2$

<proof>

lemma *in-orthogonal-complement-basis*:

fixes $B::'a::\{\text{euclidean-space}\}$ *set*

assumes *S*: *subspace* S

and *ind-B*: *independent* B

and $B: B \subseteq S$

and *span-B*: $S \subseteq \text{span } B$

shows $(v \in \text{orthogonal-complement } S) = (\forall a \in B. \text{orthogonal } a \ v)$

<proof>

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 $X::'a::\{\text{euclidean-space}\}$ *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 $(v - \text{proj-onto } v X) \in \text{orthogonal-complement } S$
 $\langle \text{proof} \rangle$

Part 2 of the Theorem 1.7 in the previous website.

lemma *UNIV-orthogonal-complement-decomposition*:
fixes $S::'a::\{\text{euclidean-space}\}$ set
assumes s : subspace S
shows $\text{UNIV} = S + (\text{orthogonal-complement } S)$
 $\langle \text{proof} \rangle$

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 $\text{norm } (\text{normalize } x) = 1$
 $\langle \text{proof} \rangle$

lemma *normalize-0*: $(\text{normalize } x = 0) = (x = 0)$
 $\langle \text{proof} \rangle$

lemma *norm-normalize-set-of-vec*:
assumes $x \neq 0$
and $x \in \text{normalize-set-of-vec } X$
shows $\text{norm } x = 1$
 $\langle \text{proof} \rangle$

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)

<proof>

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>

lemma *card-Gram-Schmidt*:

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

assumes *distinct* xs

shows card(set (*Gram-Schmidt* xs)) ≤ card (set (xs))

<proof>

lemma *orthogonal-basis-exists*:

fixes V :: (real ^b) list

assumes B: *is-basis* (set V)

and d : *distinct* V
shows $\text{vec.independent } (\text{set } (\text{Gram-Schmidt } V)) \wedge (\text{set } V) \subseteq \text{vec.span } (\text{set } (\text{Gram-Schmidt } V))$
 $\wedge (\text{card } (\text{set } (\text{Gram-Schmidt } V)) = \text{vec.dim } (\text{set } V)) \wedge \text{pairwise orthogonal } (\text{set } (\text{Gram-Schmidt } V))$
 $\langle \text{proof} \rangle$

corollary *orthogonal-basis-exists'*:

fixes $V :: (\text{real}^b)$ *list*
assumes B : *is-basis* $(\text{set } V)$
and d : *distinct* V
shows *is-basis* $(\text{set } (\text{Gram-Schmidt } V))$
 $\wedge \text{distinct } (\text{Gram-Schmidt } V) \wedge \text{pairwise orthogonal } (\text{set } (\text{Gram-Schmidt } V))$
 $\langle \text{proof} \rangle$

3.1.2 Second way

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

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

lemma *basis-orthogonal2*:

fixes $xs :: ('a :: \{\text{real-inner}\}^b)$ *list*
shows $\text{length } (\text{Gram-Schmidt2 } xs) = \text{length } (xs)$
 $\wedge \text{span } (\text{set } (\text{Gram-Schmidt2 } xs)) = \text{span } (\text{set } xs)$
 $\wedge \text{pairwise orthogonal } (\text{set } (\text{Gram-Schmidt2 } xs))$
 $\langle \text{proof} \rangle$

lemma *card-Gram-Schmidt2*:

fixes $xs :: ('a :: \{\text{real-inner}\}^b)$ *list*
assumes *distinct* xs
shows $\text{card}(\text{set } (\text{Gram-Schmidt2 } xs)) \leq \text{card } (\text{set } (xs))$
 $\langle \text{proof} \rangle$

lemma *orthogonal-basis-exists2*:

fixes $V :: (\text{real}^b)$ *list*
assumes B : *is-basis* $(\text{set } V)$
and d : *distinct* V
shows $\text{vec.independent } (\text{set } (\text{Gram-Schmidt2 } V)) \wedge (\text{set } V) \subseteq \text{vec.span } (\text{set } (\text{Gram-Schmidt2 } V))$
 $\wedge (\text{card } (\text{set } (\text{Gram-Schmidt2 } V)) = \text{vec.dim } (\text{set } V)) \wedge \text{pairwise orthogonal } (\text{set } (\text{Gram-Schmidt2 } V))$
 $\langle \text{proof} \rangle$

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} ^rows cols::{mod-type} ^rows ⇒ nat

⇒ 'a ^rows 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(λb. (if b = from-nat k then

(column b A - (∑ x∈{column i A | i. i < b}. ((column b A) · x / (x · 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 - (∑ x∈{column i A | i. i < b}. ((column b A) · x / (x · x)) *_R x))

else (column b A)) \$ a)

⟨proof⟩

lemma *Gram-Schmidt-column-k-code*[code abstract]:

vec-nth (*Gram-Schmidt-column-k* A k) = *Gram-Schmidt-column-k-row* A k

⟨proof⟩

Proofs

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

Gram-Schmidt-upt-k A (Suc k) = (*Gram-Schmidt-column-k* (*Gram-Schmidt-upt-k* A k) (Suc k))

⟨proof⟩

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

fixes A::'a::{real-inner} ^rows cols::{mod-type} ^rows

assumes *i-less-suc*: to-nat i < (Suc k)

and *suc-less-card*: Suc k < CARD ('cols)

shows column i (*Gram-Schmidt-upt-k* A (Suc k)) = column i (*Gram-Schmidt-upt-k* A k)

⟨proof⟩

lemma *column-set-Gram-Schmidt-upt-k*:
fixes $A::'a::\{\text{real-inner}\}^{\wedge'}\text{cols}::\{\text{mod-type}\}^{\wedge'}\text{rows}$
assumes $k: \text{Suc } k < \text{CARD } ('cols)$
shows $\{\text{column } i \text{ (Gram-Schmidt-upt-k } A \text{ (Suc } k)) \mid i. \text{to-nat } i \leq (\text{Suc } k)\} =$
 $\{\text{column } i \text{ (Gram-Schmidt-upt-k } A \text{ } k) \mid i. \text{to-nat } i \leq k\} \cup \{\text{column (from-nat (Suc } k)) \text{ (Gram-Schmidt-upt-k } A \text{ } k)}\}$
 $- (\sum x \in \{\text{column } i \text{ (Gram-Schmidt-upt-k } A \text{ } k) \mid i. \text{to-nat } i \leq k\}. (x \cdot (\text{column (from-nat (Suc } k)) \text{ (Gram-Schmidt-upt-k } A \text{ } k)) / (x \cdot x)) *_R x)$
 $\langle \text{proof} \rangle$

lemma *orthogonal-Gram-Schmidt-upt-k*:
assumes $s: k < \text{ncols } A$
shows *pairwise orthogonal* $(\{\text{column } i \text{ (Gram-Schmidt-upt-k } A \text{ } k) \mid i. \text{to-nat } i \leq k\})$
 $\langle \text{proof} \rangle$

lemma *columns-Gram-Schmidt-matrix-rw*:
 $\{\text{column } i \text{ (Gram-Schmidt-matrix } A) \mid i. i \in \text{UNIV}\}$
 $= \{\text{column } i \text{ (Gram-Schmidt-upt-k } A \text{ (ncols } A - 1)) \mid i. \text{to-nat } i \leq (\text{ncols } A - 1)\}$
 $\langle \text{proof} \rangle$

corollary *orthogonal-Gram-Schmidt-matrix*:
shows *pairwise orthogonal* $(\{\text{column } i \text{ (Gram-Schmidt-matrix } A) \mid i. i \in \text{UNIV}\})$
 $\langle \text{proof} \rangle$

corollary *orthogonal-Gram-Schmidt-matrix2*:
shows *pairwise orthogonal* $(\text{columns (Gram-Schmidt-matrix } A))$
 $\langle \text{proof} \rangle$

lemma *column-Gram-Schmidt-column-k*:
fixes $A::'a::\{\text{real-inner}\}^{\wedge'}n::\{\text{mod-type}\}^{\wedge'}m::\{\text{mod-type}\}$
shows $\text{column } k \text{ (Gram-Schmidt-column-k } A \text{ (to-nat } k)) =$
 $(\text{column } k \text{ } A) - (\sum x \in \{\text{column } i \text{ } A \mid i. i < k\}. (x \cdot (\text{column } k \text{ } A) / (x \cdot x)) *_R x)$
 $\langle \text{proof} \rangle$

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 $\text{column } i \text{ (Gram-Schmidt-column-k } A \text{ (to-nat } k)) = (\text{column } i \text{ } A)$
 $\langle \text{proof} \rangle$

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

lemma *cols-upt-k-insert*:

fixes $A::'a\text{ }^n::\{\text{mod-type}\}\text{ }^m::\{\text{mod-type}\}$
assumes $k: (\text{Suc } k) < \text{ncols } A$
shows $\text{cols-upt-}k\ A\ (\text{Suc } k) = (\text{insert } (\text{column } (\text{from-nat } (\text{Suc } k))\ A)\ (\text{cols-upt-}k\ A\ k))$
<proof>

lemma *columns-eq-cols-upt-k:*
fixes $A::'a\text{ }^{\text{cols}}::\{\text{mod-type}\}\text{ }^{\text{rows}}::\{\text{mod-type}\}$
shows $\text{cols-upt-}k\ A\ (\text{ncols } A - 1) = \text{columns } A$
<proof>

lemma *span-cols-upt-k-Gram-Schmidt-column-k:*
fixes $A::'a::\{\text{real-inner}\}\text{ }^n::\{\text{mod-type}\}\text{ }^m::\{\text{mod-type}\}$
assumes $k < \text{ncols } A$
and $j < \text{ncols } A$
shows $\text{span } (\text{cols-upt-}k\ A\ k) = \text{span } (\text{cols-upt-}k\ (\text{Gram-Schmidt-column-}k\ A\ j)\ k)$
<proof>

corollary *span-Gram-Schmidt-column-k:*
fixes $A::'a::\{\text{real-inner}\}\text{ }^n::\{\text{mod-type}\}\text{ }^m::\{\text{mod-type}\}$
assumes $k < \text{ncols } A$
shows $\text{span } (\text{columns } A) = \text{span } (\text{columns } (\text{Gram-Schmidt-column-}k\ A\ k))$
<proof>

corollary *span-Gram-Schmidt-upt-k:*
fixes $A::'a::\{\text{real-inner}\}\text{ }^n::\{\text{mod-type}\}\text{ }^m::\{\text{mod-type}\}$
assumes $k < \text{ncols } A$
shows $\text{span } (\text{columns } A) = \text{span } (\text{columns } (\text{Gram-Schmidt-upt-}k\ A\ k))$
<proof>

corollary *span-Gram-Schmidt-matrix:*
fixes $A::'a::\{\text{real-inner}\}\text{ }^n::\{\text{mod-type}\}\text{ }^m::\{\text{mod-type}\}$
shows $\text{span } (\text{columns } A) = \text{span } (\text{columns } (\text{Gram-Schmidt-matrix } A))$
<proof>

lemma *is-basis-columns-Gram-Schmidt-matrix:*
fixes $A::\text{real}\text{ }^n::\{\text{mod-type}\}\text{ }^m::\{\text{mod-type}\}$
assumes $b: \text{is-basis } (\text{columns } A)$
and $c: \text{card } (\text{columns } A) = \text{ncols } A$
shows $\text{is-basis } (\text{columns } (\text{Gram-Schmidt-matrix } A))$
 $\wedge \text{card } (\text{columns } (\text{Gram-Schmidt-matrix } A)) = \text{ncols } A$
<proof>

From here on, we present some lemmas that will be useful for the formalisation of the QR decomposition.

lemma *column-gr-k-Gram-Schmidt-upt*:
fixes $A::\text{real}^n::\{\text{mod-type}\}^m::\{\text{mod-type}\}$
assumes $i > k$
and $i < \text{ncols } A$
shows $\text{column } (\text{from-nat } i) (\text{Gram-Schmidt-upt-k } A \ k) = \text{column } (\text{from-nat } i) \ A$
 $\langle \text{proof} \rangle$

lemma *columns-Gram-Schmidt-upt-k-rw*:
fixes $A::\text{real}^n::\{\text{mod-type}\}^m::\{\text{mod-type}\}$
assumes $k: \text{Suc } k < \text{ncols } A$
shows $\{\text{column } i (\text{Gram-Schmidt-upt-k } A (\text{Suc } k)) \mid i. i < \text{from-nat } (\text{Suc } k)\}$
 $= \{\text{column } i (\text{Gram-Schmidt-upt-k } A \ k) \mid i. i < \text{from-nat } (\text{Suc } k)\}$
 $\langle \text{proof} \rangle$

lemma *column-Gram-Schmidt-upt-k*:
fixes $A::\text{real}^n::\{\text{mod-type}\}^m::\{\text{mod-type}\}$
assumes $k < \text{ncols } A$
shows $\text{column } (\text{from-nat } k) (\text{Gram-Schmidt-upt-k } A \ k) =$
 $(\text{column } (\text{from-nat } k) \ A) - (\sum x \in \{\text{column } i (\text{Gram-Schmidt-upt-k } A \ k) \mid i. i <$
 $(\text{from-nat } k)\}. (x \cdot (\text{column } (\text{from-nat } k) \ A) / (x \cdot x)) *_R x)$
 $\langle \text{proof} \rangle$

lemma *column-Gram-Schmidt-upt-k-preserves2*:
fixes $A::\text{real}^n::\{\text{mod-type}\}^m::\{\text{mod-type}\}$
assumes $a \leq (\text{from-nat } i)$
and $i \leq j$
and $j < \text{ncols } A$
shows $\text{column } a (\text{Gram-Schmidt-upt-k } A \ i) = \text{column } a (\text{Gram-Schmidt-upt-k } A$
 $j)$
 $\langle \text{proof} \rangle$

lemma *set-columns-Gram-Schmidt-matrix*:
fixes $A::\text{real}^n::\{\text{mod-type}\}^m::\{\text{mod-type}\}$
shows $\{\text{column } i (\text{Gram-Schmidt-matrix } A) \mid i. i < k\} = \{\text{column } i (\text{Gram-Schmidt-upt-k}$
 $A (\text{to-nat } k)) \mid i. i < k\}$
 $\langle \text{proof} \rangle$

lemma *column-Gram-Schmidt-matrix*:
fixes $A::\text{real}^n::\{\text{mod-type}\}^m::\{\text{mod-type}\}$
shows $\text{column } k (\text{Gram-Schmidt-matrix } A)$
 $= (\text{column } k \ A) - (\sum x \in \{\text{column } i (\text{Gram-Schmidt-matrix } A) \mid i. i < k\}. (x \cdot$
 $(\text{column } k \ A) / (x \cdot x)) *_R x)$

<proof>

corollary *column-Gram-Schmidt-matrix2*:

fixes $A::\text{real}^n::\{\text{mod-type}\}^m::\{\text{mod-type}\}$
shows $(\text{column } k \ A) = \text{column } k \ (\text{Gram-Schmidt-matrix } A)$
 $+ (\sum_{x \in \{\text{column } i \ (\text{Gram-Schmidt-matrix } A) \mid i. i < k\}} (x \cdot (\text{column } k \ A) / (x \cdot x))) *_R x)$
<proof>

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 $\text{vec.independent} \ (\text{columns} \ (\text{Gram-Schmidt-matrix } A)) \wedge \text{card} \ (\text{columns} \ (\text{Gram-Schmidt-matrix } A)) = \text{ncols } A$
<proof>

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 $c: \text{column } i \ (\text{Gram-Schmidt-matrix } A) = \text{column } ia \ (\text{Gram-Schmidt-matrix } A)$
shows $i = ia$
<proof>

lemma *scaleR-columns-Gram-Schmidt-matrix*:

fixes $A::\text{real}^n::\{\text{mod-type}\}^m::\{\text{mod-type}\}$
assumes $i \neq j$
and $\text{rank } A = \text{ncols } A$
shows $\text{column } j \ (\text{Gram-Schmidt-matrix } A) \cdot \text{column } i \ (\text{Gram-Schmidt-matrix } A) = 0$
<proof>

3.1.4 Examples of execution

Code lemma

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

value $let \ a = \text{map} \ (\text{list-to-vec}::\text{real list} \Rightarrow \text{real}^4) \ [[4, -2, -1, 2],$
 $[-6, 3, 4, -8], [5, -5, -3, -4]] \ in$
 $\text{map} \ \text{vec-to-list} \ (\text{Gram-Schmidt} \ a)$

value $let \ a = \text{map} \ (\text{list-to-vec}::\text{real list} \Rightarrow \text{real}^4) \ [[4, -2, -1, 2],$
 $[-6, 3, 4, -8], [5, -5, -3, -4]] \ in$
 $\text{map} \ \text{vec-to-list} \ (\text{Gram-Schmidt2} \ a)$

```

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* $Q * Q = mat\ 1$. 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:

$$(transpose A) ** A *v x = (transpose A) *v b$$

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:

$$x = (inverse R)**(transpose Q) *v b$$

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

$$(transpose R) ** R *v x = (transpose R) ** (transpose Q) *v b$$

But the R matrix is not invertible (so neither is $transpose R$). The left part of the equation $(transpose R) ** R$ 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* $A = (\chi a b. normalize (column b A) \$ a)$

Properties

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

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

assumes $a: column a A \neq 0$

shows $norm (column a (divide-by-norm A)) = 1$

<proof>

lemma *span-columns-divide-by-norm*:

shows $span (columns A) = span (columns (divide-by-norm A))$

<proof>

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]*:

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

<proof>

lemma *divide-by-norm-code* [*code abstract*]:
 $vec_nth (divide_by_norm A) = divide_by_norm_row A$
<proof>

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* $A = (let Q = divide_by_norm (Gram_Schmidt_matrix A) in (Q, (transpose Q) ** A))$

lemma *is-basis-columns-fst-QR-decomposition*:
fixes $A::real^{n::\{mod-type\}}^{m::\{mod-type\}}$
assumes $b: is_basis (columns A)$
and $c: card (columns A) = ncols A$
shows $is_basis (columns (fst (QR_decomposition A)))$
 $\wedge card (columns (fst (QR_decomposition A))) = ncols A$
<proof>

lemma *orthogonal-fst-QR-decomposition*:
shows $pairwise_orthogonal (columns (fst (QR_decomposition A)))$
<proof>

lemma *qk-uk-norm*:
 $(1/(norm (column k ((Gram_Schmidt_matrix A)))) *_R (column k ((Gram_Schmidt_matrix A))))$
 $= column k (fst(QR_decomposition A))$
<proof>

lemma *norm-columns-fst-QR-decomposition*:
fixes $A::real^{n::\{mod-type\}}^{m::\{mod-type\}}$
assumes $rank A = ncols A$
shows $norm (column i (fst (QR_decomposition A))) = 1$
<proof>

corollary *span-fst-QR-decomposition*:
fixes $A::real^{n::\{mod-type\}}^{m::\{mod-type\}}$
shows $vec.span (columns A) = vec.span (columns (fst (QR_decomposition A)))$
<proof>

corollary *col-space-QR-decomposition*:

fixes $A::\text{real}^n::\{\text{mod-type}\}^m::\{\text{mod-type}\}$
shows $\text{col-space } A = \text{col-space } (\text{fst } (\text{QR-decomposition } A))$
 $\langle \text{proof} \rangle$

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 $\text{vec.independent } (\text{columns } (\text{fst } (\text{QR-decomposition } A)))$
 $\wedge \text{card } (\text{columns } (\text{fst } (\text{QR-decomposition } A))) = \text{ncols } A$
 $\langle \text{proof} \rangle$

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 $\text{transpose } (\text{fst } (\text{QR-decomposition } A)) ** (\text{fst } (\text{QR-decomposition } A)) =$
 $\text{mat } 1$
 $\langle \text{proof} \rangle$

corollary *orthogonal-matrix-fst-QR-decomposition':*
fixes $A::\text{real}^n::\{\text{mod-type}\}^n::\{\text{mod-type}\}$
assumes $\text{rank } A = \text{ncols } A$
shows $\text{orthogonal-matrix } (\text{fst } (\text{QR-decomposition } A))$
 $\langle \text{proof} \rangle$

lemma *column-eq-fst-QR-decomposition:*
fixes $A::\text{real}^n::\{\text{mod-type}\}^m::\{\text{mod-type}\}$
assumes $r: \text{rank } A = \text{ncols } A$
and $c: \text{column } i (\text{fst } (\text{QR-decomposition } A)) = \text{column } ia (\text{fst } (\text{QR-decomposition } A))$
shows $i = ia$
 $\langle \text{proof} \rangle$

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 $\text{column } k ((\text{Gram-Schmidt-matrix } A))$
 $= (\text{column } k A) - (\sum_{x \in \{\text{column } i (\text{fst } (\text{QR-decomposition } A)) \mid i. i < k\}} (x \cdot (\text{column } k A) / (x \cdot x)) *_{\mathbb{R}} x)$
 $\langle \text{proof} \rangle$

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 $\text{column } k A = \text{column } k ((\text{Gram-Schmidt-matrix } A))$
 $+ (\sum_{x \in \{\text{column } i (\text{fst } (\text{QR-decomposition } A)) \mid i. i < k\}} (x \cdot (\text{column } k A) / (x$

• x) $*_R x$)
 ⟨proof⟩

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⟩

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 $(\text{column } k \text{ } A)$
 $= (\sum x \in \{\text{column } i \text{ (fst (QR-decomposition } A)) \mid i. i \leq k\}. (x \cdot (\text{column } k \text{ } A)) *_R$
 $x)$
 ⟨proof⟩

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

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

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⟩

lemma *R-Qi-Aj*:

fixes $A::\text{real}^n::\{\text{mod-type}\}^m::\{\text{mod-type}\}$
shows $(\text{snd (QR-decomposition } A)) \$ i \$ j = \text{column } i \text{ (fst (QR-decomposition } A)) \cdot \text{column } j \text{ } A$
 ⟨proof⟩

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⟩

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 $A = (\text{fst } (\text{QR-decomposition } A)) ** (\text{snd } (\text{QR-decomposition } A))$
 $\langle \text{proof} \rangle$

lemma *upper-triangular-snd-QR-decomposition*:
fixes $A :: \text{real}^n :: \{\text{mod-type}\}^m :: \{\text{mod-type}\}$
assumes r : $\text{rank } A = \text{ncols } A$
shows $\text{upper-triangular } (\text{snd } (\text{QR-decomposition } A))$
 $\langle \text{proof} \rangle$

lemma *upper-triangular-invertible*:
fixes $A :: \text{real}^n :: \{\text{finite, wellorder}\}^n :: \{\text{finite, wellorder}\}$
assumes u : $\text{upper-triangular } A$
and d : $\forall i. A \$ i \$ i \neq 0$
shows $\text{invertible } A$
 $\langle \text{proof} \rangle$

lemma *invertible-snd-QR-decomposition*:
fixes $A :: \text{real}^n :: \{\text{mod-type}\}^m :: \{\text{mod-type}\}$
assumes r : $\text{rank } A = \text{ncols } A$
shows $\text{invertible } (\text{snd } (\text{QR-decomposition } A))$
 $\langle \text{proof} \rangle$

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

lemma *QR-decomposition-square*:
fixes $A :: \text{real}^n :: \{\text{mod-type}\}^n :: \{\text{mod-type}\}$
assumes r : $\text{rank } A = \text{ncols } A$
shows $A = \text{fst } (\text{QR-decomposition } A) ** \text{snd } (\text{QR-decomposition } A) \wedge$
 $\text{orthogonal-matrix } (\text{fst } (\text{QR-decomposition } A)) \wedge$
 $\text{upper-triangular } (\text{snd } (\text{QR-decomposition } A)) \wedge$
 $\text{invertible } (\text{snd } (\text{QR-decomposition } A)) \wedge$

pairwise orthogonal (columns (fst (QR-decomposition A))) \wedge
($\forall i. \text{norm (column } i \text{ (fst (QR-decomposition A)))} = 1$) \wedge
vec.independent (columns (fst (QR-decomposition A))) \wedge
col-space A = col-space (fst (QR-decomposition A)) \wedge
card (columns A) = card (columns (fst (QR-decomposition A)))
 <proof>

QR for computing determinants

lemma *det-QR-decomposition:*

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

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

shows $|\det A| = |(\text{prod } (\lambda i. \text{snd}(\text{QR-decomposition } A)\$i\$i) (\text{UNIV}::'n \text{ set}))|$

<proof>

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

shows $\text{null-space } A = \text{orthogonal-complement (row-space } A)$

<proof>

lemma *left-null-space-orthogonal-complement-col-space:*

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

shows $\text{left-null-space } A = \text{orthogonal-complement (col-space } A)$

<proof>

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 $X::'a::\{\text{euclidean-space}\} \text{ set}$

assumes $\text{subspace-}S: \text{subspace } S$

and $\text{ind-}X: \text{independent } X$

and $X: X \subseteq S$

and $\text{span-}X: S \subseteq \text{span } X$
and $o: \text{pairwise orthogonal } X$
and $\text{not-eq: proj-onto } v X \neq y$
and $y: y \in S$
shows $\text{norm } (v - \text{proj-onto } v X) < \text{norm } (v - y)$
 <proof>

lemma *least-squares-approximation2*:
fixes $S::'a::\{\text{euclidean-space}\}$ set
assumes $\text{subspace-}S: \text{subspace } S$
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>

corollary *least-squares-approximation3*:
fixes $S::'a::\{\text{euclidean-space}\}$ set
assumes $\text{subspace-}S: \text{subspace } S$
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>

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>

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 $\text{subspace-}S: \text{subspace } S$
shows $\exists! p \in S. \forall y \in S - \{p\}. \text{norm } (v - p) < \text{norm } (v - y)$
 <proof>

corollary *least-squares-approximation4'*:
fixes $S::'a::\{\text{euclidean-space}\}$ set
assumes $\text{subspace-}S: \text{subspace } S$
shows $\exists! p \in S. \forall y \in S. \text{norm } (v - p) \leq \text{norm } (v - y)$
 <proof>

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

<proof>

corollary *least-squares-approximation5'*:

fixes $S::'a::\{\text{euclidean-space}\}$ *set*

assumes *subspace-S: subspace S*

shows $\exists!p \in S. \forall y \in S. \text{norm } (v - p) \leq \text{norm } (v - y) \wedge v - p \in \text{orthogonal-complement } S$

<proof>

corollary *least-squares-approximation6*:

fixes $S::'a::\{\text{euclidean-space}\}$ *set*

assumes *subspace-S: subspace S*

and $p \in S$

and $\forall y \in S. \text{norm } (v - p) \leq \text{norm } (v - y)$

shows $v - p \in \text{orthogonal-complement } S$

<proof>

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>

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>

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

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

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

<proof>

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

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

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>

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 $(x \in \text{set-least-squares-approximation } A \ b) = ((\text{snd } (\text{QR-decomposition } A))$
 $*v \ x = \text{transpose } (\text{fst } (\text{QR-decomposition } A)) *v \ b)$
 $\langle \text{proof} \rangle$

corollary *in-set-least-squares-approximation-eq-full-rank-QR2*:
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 $(x \in \text{set-least-squares-approximation } A \ b) = (x = \text{matrix-inv } (\text{snd } (\text{QR-decomposition}$
 $A)) ** \text{transpose } (\text{fst } (\text{QR-decomposition } A)) *v \ b)$
 $\langle \text{proof} \rangle$

lemma *set-least-squares-approximation-unique-solution*:
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 $(\text{set-least-squares-approximation } A \ b) = \{\text{matrix-inv } (\text{transpose } A **$
 $A)**\text{transpose } A *v \ b\}$
 $\langle \text{proof} \rangle$

lemma *set-least-squares-approximation-unique-solution-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 $(\text{set-least-squares-approximation } A \ b) = \{\text{matrix-inv } (\text{snd } (\text{QR-decomposition}$
 $A)) ** \text{transpose } (\text{fst } (\text{QR-decomposition } A)) *v \ b\}$
 $\langle \text{proof} \rangle$

end

6 Examples of execution using floats

theory *Examples-QR-Abstract-Float*
imports
QR-Decomposition
Gauss-Jordan.Examples-Gauss-Jordan-Abstract
HOL-Library.Code-Real-Approx-By-Float
begin

6.0.1 Examples

definition *example1* = $(\text{let } A = \text{list-of-list-to-matrix } [[1,2,4],[9,4,5],[0,0,0]]::\text{real}^{\wedge}3^{\wedge}3$
 in
 $\text{matrix-to-list-of-list } (\text{divide-by-norm } A))$

definition *example2* = $(\text{let } A = \text{list-of-list-to-matrix } [[1,2,4],[9,4,5],[0,0,4]]::\text{real}^{\wedge}3^{\wedge}3$
 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
  Gauss-Jordan.Examples-Gauss-Jordan-Abstract
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*]: $\text{vec-nth } (\text{show-vec-real } v) = (\% \ i. \text{show-real } (v \ \$ \ i))$
<proof>

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

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

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

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

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

value *let* $A = \text{list-of-list-to-matrix } [[1,2,1],[9,4,9]]::\text{real}^3^2$ *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]]::real³² 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* ($\lambda a. \text{if } a < \text{length-A then } A!!a \text{ else } 0$) *n*;
B' = *IArray.of-fun* ($\lambda a. \text{if } a < \text{length-B then } B!!a \text{ else } 0$) *n*
in *IArray.of-fun* ($\lambda a. A' !! a + B' !! a$) *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*)
 ⟨*proof*⟩

lemma *of-fun-nth*:
assumes *i: i < n*
shows (*IArray.of-fun f n*) !! *i = f i*
 ⟨*proof*⟩

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*⟩
end

instantiation *iarray* :: (*zero*) *zero*
begin
definition *zero-iarray* = (*IArray[]::'a iarray*)
instance ⟨*proof*⟩
end

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

instance
 ⟨*proof*⟩
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*⟩
end

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

```

definition minus-iarray :: 'a iarray  $\Rightarrow$  'a iarray  $\Rightarrow$  '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 ( $\lambda a$ . if a < length-A then A!!a else 0) n;
        B'=IArray.of-fun ( $\lambda a$ . if a < length-B then B!!a else 0) n
    in
    IArray.of-fun ( $\lambda a$ . A' !! a - B' !! a) n)

```

```

instance <proof>
end

```

8.2 Some previous definitions and properties for IArrays

8.2.1 Lemmas

8.2.2 Definitions

```

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

```

```

fun exists :: ('a  $\Rightarrow$  bool)  $\Rightarrow$  'a iarray  $\Rightarrow$  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  $\rightarrow$  (SML) Vector.exists
| constant IArray-Addenda-QR.all  $\rightarrow$  (SML) Vector.all

```

```

end

```

9 Matrices as nested IArrays

```

theory Matrix-To-IArray-QR

```

```

imports

```

```

  Gauss-Jordan.Elementary-Operations
  Rank-Nullity-Theorem.Mod-Type
  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)
<proof>

lemma *vec-to-iarray-nth'*:
fixes A::'aⁿ::{mod-type}
shows (vec-to-iarray A) !! (to-nat i) = A \$ i
<proof>

lemma *iarray-to-vec-nth*:
shows (iarray-to-vec A) \$ i = A !! (to-nat i)
<proof>

lemma *vec-to-iarray-morph*:
fixes A::'aⁿ::{mod-type}
shows (A = B) = (vec-to-iarray A = vec-to-iarray B)
<proof>

lemma *inj-vec-to-iarray*:
shows inj vec-to-iarray
<proof>

lemma *iarray-to-vec-vec-to-iarray*:
fixes A::'aⁿ::{mod-type}
shows iarray-to-vec (vec-to-iarray A) = A
<proof>

lemma *vec-to-iarray-iarray-to-vec*:
assumes length-eq: IArray.length A = CARD('n::{mod-type})
shows vec-to-iarray (iarray-to-vec A::'aⁿ::{mod-type}) = A
<proof>

lemma *length-vec-to-iarray*:
fixes xa::'aⁿ::{mod-type}

shows $IArray.length (vec\text{-}to\text{-}iarray\ x) = CARD('n)$
 $\langle proof \rangle$

9.1.2 Isomorphism between matrix and nested iarrays

definition $matrix\text{-}to\text{-}iarray :: 'a \sim n :: \{mod\text{-}type\} \sim m :: \{mod\text{-}type\} \Rightarrow 'a\ iarray\ iarray$

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

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

lemma $matrix\text{-}to\text{-}iarray\text{-}morph:$

fixes $A :: 'a \sim n :: \{mod\text{-}type\} \sim m :: \{mod\text{-}type\}$

shows $(A = B) = (matrix\text{-}to\text{-}iarray\ A = matrix\text{-}to\text{-}iarray\ B)$

$\langle proof \rangle$

lemma $matrix\text{-}to\text{-}iarray\text{-}eq\text{-}of\text{-}fun:$

fixes $A :: 'a \sim columns :: \{mod\text{-}type\} \sim rows :: \{mod\text{-}type\}$

assumes $vec\text{-}eq\text{-}f: \forall i.\ vec\text{-}to\text{-}iarray\ (A\ \$\ i) = f\ (to\text{-}nat\ i)$

and $n\text{-}eq\text{-}length: n = IArray.length (matrix\text{-}to\text{-}iarray\ A)$

shows $matrix\text{-}to\text{-}iarray\ A = IArray.of\text{-}fun\ f\ n$

$\langle proof \rangle$

lemma $map\text{-}vec\text{-}to\text{-}iarray\text{-}rw[simp]:$

fixes $A :: 'a \sim columns :: \{mod\text{-}type\} \sim rows :: \{mod\text{-}type\}$

shows $map (\lambda x.\ vec\text{-}to\text{-}iarray\ (A\ \$\ from\text{-}nat\ x)) [0..<CARD('rows)] !\ to\text{-}nat\ i = vec\text{-}to\text{-}iarray\ (A\ \$\ i)$

$\langle proof \rangle$

lemma $matrix\text{-}to\text{-}iarray\text{-}nth:$

$matrix\text{-}to\text{-}iarray\ A\ !!\ to\text{-}nat\ i\ !!\ to\text{-}nat\ j = A\ \$\ i\ \$\ j$

$\langle proof \rangle$

lemma $vec\text{-}matrix: vec\text{-}to\text{-}iarray\ (A\ \$\ i) = (matrix\text{-}to\text{-}iarray\ A)\ !!\ (to\text{-}nat\ i)$

$\langle proof \rangle$

lemma $iarray\text{-}to\text{-}matrix\text{-}matrix\text{-}to\text{-}iarray:$

fixes $A :: 'a \sim columns :: \{mod\text{-}type\} \sim rows :: \{mod\text{-}type\}$

shows $iarray\text{-}to\text{-}matrix\ (matrix\text{-}to\text{-}iarray\ A) = A$

$\langle proof \rangle$

9.2 Definition of operations over matrices implemented by iarrays

definition $mult\text{-}iarray :: 'a :: \{times\}\ iarray \Rightarrow 'a \Rightarrow 'a\ iarray$

where $mult\text{-}iarray\ A\ q = IArray.of\text{-}fun (\lambda n.\ q * A!!n) (IArray.length\ A)$

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

definition *column-iarray* :: nat => 'a iarray iarray => 'a iarray
 where *column-iarray* k A = IArray.of-fun (λm. A !! m !! k) (IArray.length A)

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

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

definition *rows-iarray* A = {row-iarray i A | i. i ∈ {..*nrows-iarray* A}}

definition *columns-iarray* A = {column-iarray i A | i. i ∈ {..*ncols-iarray* A}}

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

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

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

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

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

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

definition *is-zero-iarray* :: 'a::{zero} iarray => bool
 where *is-zero-iarray* A = IArray-Addenda-QR.all (λi. A !! i = 0) (IArray[0..*IArray.length*
 A])

9.2.1 Properties of previous definitions

lemma *is-zero-iarray-eq-iff*:

fixes A::'a::{zero} ~n::{mod-type}

shows (A = 0) = (*is-zero-iarray* (*vec-to-iarray* A))

<proof>

lemma *mult-iarray-works*:

assumes $a < IArray.length\ A$ **shows** $mult\text{-}iarray\ A\ q\ !!\ a = q * A !! a$

<proof>

lemma *length-eq-card-rows*:

fixes $A :: 'a\ ^\prime\ columns :: \{mod\text{-}type\}^\prime\ rows :: \{mod\text{-}type\}$

shows $IArray.length\ (matrix\text{-}to\text{-}iarray\ A) = CARD('rows)$

<proof>

lemma *nrows-eq-card-rows*:

fixes $A :: 'a\ ^\prime\ columns :: \{mod\text{-}type\}^\prime\ rows :: \{mod\text{-}type\}$

shows $nrows\text{-}iarray\ (matrix\text{-}to\text{-}iarray\ A) = CARD('rows)$

<proof>

lemma *length-eq-card-columns*:

fixes $A :: 'a\ ^\prime\ columns :: \{mod\text{-}type\}^\prime\ rows :: \{mod\text{-}type\}$

shows $IArray.length\ (matrix\text{-}to\text{-}iarray\ A\ !!\ 0) = CARD('columns)$

<proof>

lemma *ncols-eq-card-columns*:

fixes $A :: 'a\ ^\prime\ columns :: \{mod\text{-}type\}^\prime\ rows :: \{mod\text{-}type\}$

shows $ncols\text{-}iarray\ (matrix\text{-}to\text{-}iarray\ A) = CARD('columns)$

<proof>

lemma *matrix-to-iarray-nrows*:

fixes $A :: 'a\ ^\prime\ columns :: \{mod\text{-}type\}^\prime\ rows :: \{mod\text{-}type\}$

shows $nrows\ A = nrows\text{-}iarray\ (matrix\text{-}to\text{-}iarray\ A)$

<proof>

lemma *matrix-to-iarray-ncols*:

fixes $A :: 'a\ ^\prime\ columns :: \{mod\text{-}type\}^\prime\ rows :: \{mod\text{-}type\}$

shows $ncols\ A = ncols\text{-}iarray\ (matrix\text{-}to\text{-}iarray\ A)$

<proof>

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

<proof>

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

<proof>

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

<proof>

lemma *vec-to-iarray-column'*:

assumes $k: k < n\text{cols } A$
shows $(\text{vec-to-iarray } (\text{column } (\text{from-nat } k) A)) = (\text{column-iarray } k (\text{matrix-to-iarray } A))$
 $\langle \text{proof} \rangle$

lemma *column-iarray-nth*:
assumes $i: i < n\text{rows-iarray } A$
shows $\text{column-iarray } j A !! i = A !! i !! j$
 $\langle \text{proof} \rangle$

lemma *vec-to-iarray-rows*: $\text{vec-to-iarray}' (\text{rows } A) = \text{rows-iarray } (\text{matrix-to-iarray } A)$
 $\langle \text{proof} \rangle$

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

9.3 Definition of elementary operations

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

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

definition *mult-row-iarray* :: $'a::\{\text{times}\} \text{ iarray iarray} \Rightarrow \text{nat} \Rightarrow 'a \Rightarrow 'a \text{ iarray iarray}$

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

definition *row-add-iarray* :: $'a::\{\text{plus, times, zero}\} \text{ iarray iarray} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow 'a \Rightarrow 'a \text{ iarray iarray}$

where *row-add-iarray* $A a b q = \text{IArray.of-fun } (\lambda n. \text{if } n=a \text{ then } A!!a + \text{mult-iarray } (A!!b) q \text{ else } A!!n) (\text{IArray.length } A)$

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

where *interchange-columns-iarray* $A a b = \text{tabulate2 } (n\text{rows-iarray } A) (n\text{cols-iarray } A) (\lambda i j. \text{if } j = a \text{ then } A !! i !! b \text{ else if } j = b \text{ then } A !! i !! a \text{ else } A !! i !! j)$

definition *mult-column-iarray* :: $'a::\{\text{times}\} \text{ iarray iarray} \Rightarrow \text{nat} \Rightarrow 'a \Rightarrow 'a \text{ iarray iarray}$

where *mult-column-iarray* $A n q = \text{tabulate2 } (n\text{rows-iarray } A) (n\text{cols-iarray } A) (\lambda i j. \text{if } j = n \text{ then } A !! i !! j * q \text{ else } A !! i !! j)$

definition *column-add-iarray* :: $'a::\{\text{plus, times}\} \text{ iarray iarray} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow 'a \Rightarrow 'a \text{ iarray iarray}$

where *column-add-iarray* $A n m q = \text{tabulate2 } (n\text{rows-iarray } A) (n\text{cols-iarray } A) (\lambda i j. \text{if } j = n \text{ then } A !! i !! n + A !! i !! m * q \text{ else } A !! i !! j)$

9.3.1 Code generator

lemma *vec-to-iarray-plus*[code-unfold]: $\text{vec-to-iarray } (a + b) = (\text{vec-to-iarray } a) + (\text{vec-to-iarray } b)$
 ⟨proof⟩

lemma *matrix-to-iarray-plus*[code-unfold]: $\text{matrix-to-iarray } (A + B) = (\text{matrix-to-iarray } A) + (\text{matrix-to-iarray } B)$
 ⟨proof⟩

lemma *matrix-to-iarray-mat*[code-unfold]:
 $\text{matrix-to-iarray } (\text{mat } k :: 'a::\{\text{zero}\} \wedge 'n::\{\text{mod-type}\} \wedge 'm::\{\text{mod-type}\}) = \text{mat-iarray } k \text{ CARD}('n::\{\text{mod-type}\})$
 ⟨proof⟩

lemma *matrix-to-iarray-transpose*[code-unfold]:
shows $\text{matrix-to-iarray } (\text{transpose } A) = \text{transpose-iarray } (\text{matrix-to-iarray } A)$
 ⟨proof⟩

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 $\text{matrix-to-iarray } (A ** B) = (\text{matrix-to-iarray } A) ** i (\text{matrix-to-iarray } B)$
 ⟨proof⟩

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 $\text{vec-to-iarray } (A * v x) = (\text{matrix-to-iarray } A) * i v (\text{vec-to-iarray } x)$
 ⟨proof⟩

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 $\text{vec-to-iarray } (x v * A) = (\text{vec-to-iarray } x) v * i (\text{matrix-to-iarray } A)$
 ⟨proof⟩

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 $\text{matrix-to-iarray } (\text{interchange-rows } A \ i \ j) = \text{interchange-rows-iarray } (\text{matrix-to-iarray } A) \ (to\text{-nat } i) \ (to\text{-nat } j)$
 ⟨proof⟩

lemma *matrix-to-iarray-mult-row*[code-unfold]:
fixes $A::'a::\{\text{semiring-1}\} \wedge \text{columns}::\{\text{mod-type}\} \wedge \text{rows}::\{\text{mod-type}\}$
shows $\text{matrix-to-iarray } (\text{mult-row } A \ i \ q) = \text{mult-row-iarray } (\text{matrix-to-iarray } A) \ (to\text{-nat } i) \ q$
 ⟨proof⟩

```

lemma matrix-to-iarray-row-add[code-unfold]:
  fixes  $A::'a::\{\text{semiring-1}\}^{\wedge}\text{columns}::\{\text{mod-type}\}^{\wedge}\text{rows}::\{\text{mod-type}\}$ 
  shows  $\text{matrix-to-iarray} (\text{row-add } A \ i \ j \ q) = \text{row-add-iarray} (\text{matrix-to-iarray } A)$ 
   $(\text{to-nat } i) (\text{to-nat } j) \ q$ 
   $\langle \text{proof} \rangle$ 

lemma matrix-to-iarray-interchange-columns[code-unfold]:
  fixes  $A::'a::\{\text{semiring-1}\}^{\wedge}\text{columns}::\{\text{mod-type}\}^{\wedge}\text{rows}::\{\text{mod-type}\}$ 
  shows  $\text{matrix-to-iarray} (\text{interchange-columns } A \ i \ j) = \text{interchange-columns-iarray}$ 
   $(\text{matrix-to-iarray } A) (\text{to-nat } i) (\text{to-nat } j)$ 
   $\langle \text{proof} \rangle$ 

lemma matrix-to-iarray-mult-columns[code-unfold]:
  fixes  $A::'a::\{\text{semiring-1}\}^{\wedge}\text{columns}::\{\text{mod-type}\}^{\wedge}\text{rows}::\{\text{mod-type}\}$ 
  shows  $\text{matrix-to-iarray} (\text{mult-column } A \ i \ q) = \text{mult-column-iarray} (\text{matrix-to-iarray}$ 
   $A) (\text{to-nat } i) \ q$ 
   $\langle \text{proof} \rangle$ 

lemma matrix-to-iarray-column-add[code-unfold]:
  fixes  $A::'a::\{\text{semiring-1}\}^{\wedge}\text{columns}::\{\text{mod-type}\}^{\wedge}\text{rows}::\{\text{mod-type}\}$ 
  shows  $\text{matrix-to-iarray} (\text{column-add } A \ i \ j \ q) = \text{column-add-iarray} (\text{matrix-to-iarray}$ 
   $A) (\text{to-nat } i) (\text{to-nat } j) \ q$ 
   $\langle \text{proof} \rangle$ 

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 ::  $'a \ \text{iarray} \ \text{iarray} \Rightarrow 'a \ \text{list} \ \text{list}$ 
  where  $\text{iarray-of-iarray-to-list-of-list } A = \text{map } \text{IArray.list-of} (\text{map } (!!) \ A) [0..<\text{IArray.length}$ 
   $A]$ 

instantiation iarray ::  $(\text{scaleR}) \ \text{scaleR}$ 
begin
definition scaleR-iarray  $k \ A = \text{IArray.of-fun } (\lambda i. \ k \ *_R \ (A \ !! \ i)) (\text{IArray.length } A)$ 
instance  $\langle \text{proof} \rangle$ 
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 $\langle proof \rangle$
end

lemma *plus-iarray-component*:
assumes $iA: i < IArray.length\ A$
and $iB: i < IArray.length\ B$
shows $(A+B)!!i = A!!i + B!!i$
 $\langle proof \rangle$

lemma *minus-iarray-component*:
assumes $iA: i < IArray.length\ A$
and $iB: i < IArray.length\ B$
shows $(A-B)!!i = A!!i - B!!i$
 $\langle proof \rangle$

lemma *length-plus-iarray*:
 $IArray.length\ (A+B) = \max\ (IArray.length\ A)\ (IArray.length\ B)$
 $\langle proof \rangle$

lemma *length-sum-iarray*:
assumes *finite* S **and** $S \neq \{\}$
shows $IArray.length\ (sum\ f\ S) = \text{Max}\ \{IArray.length\ (f\ x) \mid x. x \in S\}$
 $\langle proof \rangle$

lemma *sum-component-iarray*:
assumes $a: \forall x \in S. i < IArray.length\ (f\ x)$
and $f: \text{finite}\ S$
and $S: S \neq \{\}$ — If S is empty, then the sum will return the empty iarray and it makes no sense to access the component i
shows $sum\ f\ S!!i = (\sum x \in S. f\ x!!i)$
 $\langle proof \rangle$

lemma *length-zero-iarray*: $IArray.length\ 0 = 0$
 $\langle proof \rangle$

lemma *minus-zero-iarray*:
fixes $A::'a::\{\text{group-add}\}\ iarray$
shows $A - 0 = A$
 $\langle proof \rangle$

10.2 Inner mult over real iarrays

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

lemma *vec-to-iarray-inner*:
 $a \cdot b = \text{vec-to-iarray } a \cdot i \text{ vec-to-iarray } b$
 $\langle \text{proof} \rangle$

lemma *vec-to-iarray-scaleR*:
 $\text{vec-to-iarray } (a *_R x) = a *_R (\text{vec-to-iarray } x)$
 $\langle \text{proof} \rangle$

10.3 Gram Schmidt over IArrays

definition *Gram-Schmidt-column-k-iarrays* *A k*
= *tabulate2* (*nrows-iarray A*) (*ncols-iarray A*) ($\lambda a b. (\text{if } b = k$
then (*column-iarray* *b A* - *sum* ($\lambda x. (((\text{column-iarray } b A) \cdot i x) / (x \cdot i x)) *_R$
x)

(*set* (*List.map* ($\lambda n. \text{column-iarray } n A$) [0..*b*]))))
else (*column-iarray* *b A*)) !! *a*)

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

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

lemma *matrix-to-iarray-Gram-Schmidt-column-k*:
fixes *A*::*real*[^]*cols*::{*mod-type*}[^]*rows*::{*mod-type*}
assumes *k*: *k* < *ncols A*
shows *matrix-to-iarray* (*Gram-Schmidt-column-k A k*) = *Gram-Schmidt-column-k-iarrays*
(*matrix-to-iarray A*) *k*
 $\langle \text{proof} \rangle$

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*
 $\langle \text{proof} \rangle$

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*)
 $\langle \text{proof} \rangle$

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 = sqrt (A · i A)
```

```
definition divide-by-norm-iarray A = tabulate2 (nrows-iarray A) (ncols-iarray
A)
  (λa b. ((1/norm-iarray (column-iarray b A)) *R (column-iarray b A)) !! a)
```

```
definition QR-decomposition-iarrays A = (let Q = divide-by-norm-iarray (Gram-Schmidt-matrix-iarrays
A)
  in (Q, transpose-iarray Q **i A))
```

```
lemma vec-to-iarray-norm[code-unfold]:
  shows (norm A) = norm-iarray (vec-to-iarray A)
  ⟨proof⟩
```

```
lemma matrix-to-iarray-divide-by-norm[code-unfold]:
  fixes A::real^cols::{mod-type}^rows::{mod-type}
  shows matrix-to-iarray (divide-by-norm A) = divide-by-norm-iarray (matrix-to-iarray
A)
  ⟨proof⟩
```

```
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⟩
```

```
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>

definition *matrix-to-iarray-pair* $X = (\text{matrix-to-iarray } (\text{fst } X), \text{matrix-to-iarray } (\text{snd } X))$

lemma *matrix-to-iarray-QR-decomposition*`[code-unfold]`:

shows *matrix-to-iarray-pair* (*QR-decomposition* A) = *QR-decomposition-iarrays* (*matrix-to-iarray* A)

<proof>

end

12 Examples of execution using floats and IArrays

theory *Examples-QR-IArrays-Float*

imports

QR-Decomposition-IArrays

Gauss-Jordan.Examples-Gauss-Jordan-Abstract

HOL-Library.Code-Real-Approx-By-Float

begin

12.1 Examples

definition *example1* = (let $A = \text{list-of-list-to-matrix } [[1,2,4],[9,4,5],[0,0,0]]::\text{real}^3^3$ in

iarray-of-iarray-to-list-of-list (*matrix-to-iarray* (*divide-by-norm* A)))

definition *example2* = (let $A = \text{list-of-list-to-matrix } [[1,2,4],[9,4,5],[0,0,4]]::\text{real}^3^3$ in

iarray-of-iarray-to-list-of-list (*matrix-to-iarray* (*fst* (*QR-decomposition* A))))

definition *example3* = (let $A = \text{list-of-list-to-matrix } [[1,2,4],[9,4,5],[0,0,4]]::\text{real}^3^3$ in

iarray-of-iarray-to-list-of-list (*matrix-to-iarray* (*snd* (*QR-decomposition* A))))

definition *example4* = (let $A = \text{list-of-list-to-matrix } [[1,2,4],[9,4,5],[0,0,4]]::\text{real}^3^3$ in

iarray-of-iarray-to-list-of-list (*matrix-to-iarray* (*fst* (*QR-decomposition* A) ** (*snd* (*QR-decomposition* A))))

definition *example5* = (let $A = \text{list-of-list-to-matrix } [[1,\text{sqrt } 2,4],[\text{sqrt } 5,4,5],[0,\text{sqrt } 7,4]]::\text{real}^3^3$ in

iarray-of-iarray-to-list-of-list (*matrix-to-iarray* (*fst* (*QR-decomposition* A))))

definition *example6* = (let $A = \text{list-of-list-to-matrix } [[1,\text{sqrt } 2,4],[\text{sqrt } 5,4,5],[0,\text{sqrt } 7,4]]::\text{real}^3^3$ 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* $v = IArray.of-fun (\lambda i. show-real (v !! i)) (IArray.length v)$

lemma *vec-to-iarray-show-vec-real*[code-unfold]: $vec-to-iarray (show-vec-real v) = show-vec-real-iarrays (vec-to-iarray v)$
(proof)

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* $= IArray.list-of \circ show-vec-real-iarrays \circ vec-to-iarray$

definition *show-matrix-real-iarrays* $A = IArray.of-fun (\lambda i. show-vec-real-iarrays (A !! i)) (IArray.length A)$

lemma *matrix-to-iarray-show-matrix-real*[code-unfold]: $matrix-to-iarray (show-matrix-real v) = show-matrix-real-iarrays (matrix-to-iarray v)$
(proof)

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* $A = IArray.of-fun (\lambda i. print-vec-mat (A !! i)) (IArray.length A)$

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

13.1.1 Examples

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

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

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

value *let* $A = list-of-list-to-matrix [[1,2,4],[9,4,5],[0,0,4]]::real^3^3$ 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]]::*real*³⁴ *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]]::*real*³⁴;
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*³⁴;
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*³⁴;
b = *list-to-vec* [1,2,3,4]::*real*⁴
in *matrix-to-list-of-list* (*show-matrix-real* ((*snd* (*QR-decomposition* *A*))))

least squares solution

definition *A* ≡ *list-of-list-to-matrix* [[1,3/5,3],[9,4,5/3],[0,0,4],[1,2,3]]::*real*³⁴
definition *b* ≡ *list-to-vec* [1,2,3,4]::*real*⁴

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

A times least squares solution

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

The matrix Q

value *print-mat* (*fst* (*QR-decomposition* *A*))

The matrix R

value *print-mat* (*snd* (*QR-decomposition* *A*))

The inverse of matrix R

value *let* *R* = *snd* (*QR-decomposition* *A*) *in* *print-mat* (*the* (*inverse-matrix* *R*))

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

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

value *let* *A* = *list-of-list-to-matrix* [[1,2,4],[9,4,5],[0,0,4],[3,5,4]]::*real*³⁴ *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 = IArray[IArray[1,2,4],IArray[9,4,5::real],IArray[0,0,0]] in
  iarray-of-iarray-to-list-of-list (show-matrix-real-iarrays (divide-by-norm-iarray
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)))

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 (snd (QR-decomposition-iarrays
A)))

value let A = list-of-list-to-matrix [[1,2,3],[9,4,5],[0,0,4],[1,2,3]]::real34 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]]::real34;
  b = list-to-vec [1,2,3,4]::real4 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]]::real34;
  b = list-to-vec [1,2,3,4]::real4
  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]]::real34;
  b = list-to-vec [1,2,3,4]::real4
  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]]::real34;
  b = list-to-vec [1,2,3,4]::real4;
  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⟩

```

```

lemma cnj-0[simp]: cnj 0 = 0
  ⟨proof⟩

```

```

lemma cnj-0-eq[simp]: (cnj a = 0) = (a = 0)
  ⟨proof⟩

```

```

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

```

```

    <proof>

end

lemma cnj-sum:  $cnj (\sum xa \in A. ((f xa))) = (\sum xa \in A. cnj (f xa))$ 
    <proof>

instantiation real :: cnj
begin

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

instance
    <proof>
end

instantiation complex :: cnj
begin

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

instance
    <proof>
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>

lemma real-0[simp]:  $real-of 0 = 0$ 
    <proof>

lemma real-sum:
     $real-of (sum (\lambda i. f i) A) = sum (\lambda i. real-of (f i)) A$ 
    <proof>

end

```

instantiation *real* :: *real-of-extended*
begin

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

instance
 \langle *proof* \rangle
end

instantiation *complex* :: *real-of-extended*
begin

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

instance
 \langle *proof* \rangle
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*: $0 <$ *real-of* (*inner* *x* *x*) \longleftrightarrow *x* \neq 0

interpretation *RV-inner*: *inner-product-space scaleR inner*
 \langle *proof* \rangle

interpretation *RR-inner*: *inner-product-space scaleR (*)*
 \langle *proof* \rangle

interpretation *CC-inner*: *inner-product-space ((*)::complex \Rightarrow complex \Rightarrow complex)*
 λ *x y. x*cnj y*
 \langle *proof* \rangle

context *inner-product-space*
begin

lemma *inner-zero-left* [simp]: $\text{inner } 0 \ x = 0$
<proof>

lemma *inner-minus-left* [simp]: $\text{inner } (- \ x) \ y = - \ \text{inner } \ x \ y$
<proof>

lemma *inner-diff-left*: $\text{inner } (x - y) \ z = \text{inner } \ x \ z - \text{inner } \ y \ z$
<proof>

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

Transfer distributivity rules to right argument.

lemma *inner-add-right*: $\text{inner } \ x \ (y + z) = \text{inner } \ x \ y + \text{inner } \ x \ z$
<proof>

lemma *inner-scaleR-right* [simp]: $\text{inner } \ x \ (\text{scale } \ r \ y) = (\text{cnj } \ r) * (\text{inner } \ x \ y)$
<proof>

lemma *inner-zero-right* [simp]: $\text{inner } \ x \ 0 = 0$
<proof>

lemma *inner-minus-right* [simp]: $\text{inner } \ x \ (- \ y) = - \ \text{inner } \ x \ y$
<proof>

lemma *inner-diff-right*: $\text{inner } \ x \ (y - z) = \text{inner } \ x \ y - \text{inner } \ x \ z$
<proof>

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

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

<proof>

lemma *real-inner-inner*: $\text{real-of } (\text{inner } x \ x * \text{inner } y \ y) = \text{real-of } (\text{inner } x \ x) * \text{real-of } (\text{inner } y \ y)$
<proof>

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>
end

hide-const (**open**) *norm*

context *inner-product-space*
begin

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

lemmas $\text{norm-eq-sqrt-inner} = \text{norm-def}$

lemma *inner-cnj-ge-zero[simp]*: $\text{real-of } ((\text{inner } x \ y) * \text{cnj } (\text{inner } x \ y)) \geq 0$
<proof>

lemma *power2-norm-eq-inner*: $(\text{norm } x)^2 = \text{real-of } (\text{inner } x \ x)$
<proof>

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>

end

14.3.2 Orthogonality

hide-const (**open**) *orthogonal*

context *inner-product-space*
begin

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

lemma *orthogonal-clauses*:
 $\text{orthogonal } a \ 0$
 $\text{orthogonal } a \ x \implies \text{orthogonal } a \ (\text{scale } c \ x)$
 $\text{orthogonal } a \ x \implies \text{orthogonal } a \ (-x)$
 $\text{orthogonal } a \ x \implies \text{orthogonal } a \ y \implies \text{orthogonal } a \ (x + y)$
 $\text{orthogonal } a \ x \implies \text{orthogonal } a \ y \implies \text{orthogonal } a \ (x - y)$
 $\text{orthogonal } 0 \ a$
 $\text{orthogonal } x \ a \implies \text{orthogonal } (\text{scale } c \ x) \ a$

$orthogonal\ x\ a \implies orthogonal\ (-\ x)\ a$
 $orthogonal\ x\ a \implies orthogonal\ y\ a \implies orthogonal\ (x + y)\ a$
 $orthogonal\ x\ a \implies orthogonal\ y\ a \implies orthogonal\ (x - y)\ a$
 <proof>

lemma *inner-commute-zero*: $(inner\ x\ a\ x = 0) = (inner\ x\ x\ a = 0)$
 <proof>

lemma *vector-sub-project-orthogonal*:
 $inner\ b\ (x - scale\ (inner\ x\ b / (inner\ b\ b))\ b) = 0$
 <proof>

lemma *orthogonal-commute*: $orthogonal\ x\ y \longleftrightarrow orthogonal\ y\ x$
 <proof>

lemma *pairwise-orthogonal-insert*:
assumes *pairwise orthogonal S*
and $\bigwedge y. y \in S \implies orthogonal\ x\ y$
shows *pairwise orthogonal (insert x S)*
 <proof>

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 $\forall a \in A. f\ a = 0$
 <proof>

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* :: $'a \Rightarrow^n \Rightarrow 'a \Rightarrow^n \Rightarrow 'a$
assumes *inner-vec-def*: $inner\ x\ y = sum\ (\lambda i. inner\text{-field}\ (x\$i)\ (y\$i))\ UNIV$
begin

lemma *inner-ge-zero [simp]*: $0 \leq real\text{-of}\ (inner\ x\ x)$
 <proof>

lemma *real-scalar-mult2*: $real\text{-of}\ (inner\ x\ x) *_{\mathbb{R}} A = inner\ x\ x * A$
 <proof>

lemma *i1*: $inner\ x\ y = cnj\ (inner\ y\ x)$
 <proof>

lemma *i2*: $inner\ (x + y)\ z = inner\ x\ z + inner\ y\ z$
 <proof>

lemma *i3*: *inner* (*r * s x*) *y* = *r * inner x y*
 ⟨*proof*⟩

lemma *i4*: **assumes** *inner x x = 0*
shows *x = 0*
 ⟨*proof*⟩

lemma *inner-0-0[simp]*: *inner 0 0 = 0*
 ⟨*proof*⟩

sublocale *v?*: *inner-product-space* ((**s*) :: '*a* ⇒ '*a*[^]*n* ⇒ '*a*[^]*n*) *inner*
 ⟨*proof*⟩
end

14.5 Matrices and inner product

locale *matrix* =
COLS?: *vec-real-inner* $\lambda x y. x * cnj y$ *inner-cols*
 + *ROWS?*: *vec-real-inner* $\lambda x y. x * cnj y$ *inner-rows*
for *inner-cols* :: '*a*[^]*cols*::{*finite*, *wellorder*} ⇒ '*a*[^]*cols*::{*finite*, *wellorder*} ⇒
'*a*::{*field*, *cnj*, *real-of-extended*}
and *inner-rows* :: '*a*[^]*rows*::{*finite*, *wellorder*} ⇒ '*a*[^]*rows*::{*finite*, *wellorder*} ⇒
'*a*
begin

lemma *dot-lmul-matrix*: *inner-rows* (*x v* A*) *y* = *inner-cols x* ((χ *i j. cnj* (*A* \$ *i*
 \$ *j*)) **v y*)
 ⟨*proof*⟩

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*)
 ⟨*proof*⟩

lemma *orthogonal-complement-mono*:
assumes *A-in-B*: $A \subseteq B$
shows *orthogonal-complement B* \subseteq *orthogonal-complement A*
 ⟨*proof*⟩

lemma *B-in-orthogonal-complement-of-orthogonal-complement*:

shows $B \subseteq \text{orthogonal-complement} (\text{orthogonal-complement } B)$
 ⟨proof⟩

end

14.7 Generalizing projections

context *inner-product-space*

begin

Projection of two vectors: v onto u

definition $\text{proj } v \ u = \text{scale } (\text{inner } v \ u / \text{inner } u \ u) \ u$

Projection of a onto S

definition $\text{proj-onto } a \ S = (\text{sum } (\lambda x. \text{proj } a \ x) \ S)$

lemma *vector-sub-project-orthogonal-proj*:

shows $\text{inner } b \ (x - \text{proj } x \ b) = 0$

⟨proof⟩

lemma *orthogonal-proj-set*:

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

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

⟨proof⟩

lemma *pairwise-orthogonal-proj-set*:

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

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

⟨proof⟩

end

lemma *orthogonal-real-eq*: $\text{RV-inner.orthogonal} = \text{real-inner-class.orthogonal}$

⟨proof⟩

14.8 Second Part of the Fundamental Theorem of Linear Algebra generalized

context *matrix*

begin

lemma *cnj-cnj-matrix[simp]*: $(\chi \ i \ j. \text{cnj } ((\chi \ i \ j. \text{cnj } (A \ \$ \ i \ \$ \ j)) \ \$ \ i \ \$ \ j)) = A$

⟨proof⟩

lemma *cnj-transpose[simp]*: $(\chi \ i \ j. \text{cnj } (\text{transpose } A \ \$ \ i \ \$ \ j)) = \text{transpose } (\chi \ i \ j. \text{cnj } (A \ \$ \ i \ \$ \ j))$

⟨proof⟩

lemma *null-space-orthogonal-complement-row-space*:

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

shows $\text{null-space } A = \text{COLS.v.orthogonal-complement } (\text{row-space } (\chi \ i \ j. \text{cnj } (A \ \$ \ i \ \$ \ j)))$
 $\langle \text{proof} \rangle$

lemma *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 $\text{left-null-space } A = \text{ROWS.v.orthogonal-complement } (\text{col-space } (\chi \ i \ j. \text{cnj } (A \ \$ \ i \ \$ \ j)))$
 $\langle \text{proof} \rangle$

end

We can get the explicit results for complex and real matrices

interpretation *real-matrix: matrix $\lambda x \ y::\text{real} \wedge \text{cols}::\{\text{finite}, \text{wellorder}\}$.*
 $\text{sum } (\lambda i. (x\$i) * (y\$i)) \text{ UNIV } \lambda x \ y. \text{sum } (\lambda i. (x\$i) * (y\$i)) \text{ UNIV}$
 $\langle \text{proof} \rangle$

interpretation *complex-matrix: matrix $\lambda x \ y::\text{complex} \wedge \text{cols}::\{\text{finite}, \text{wellorder}\}$.*
 $\text{sum } (\lambda i. (x\$i) * \text{cnj } (y\$i)) \text{ UNIV } \lambda x \ y. \text{sum } (\lambda i. (x\$i) * \text{cnj } (y\$i)) \text{ UNIV}$
 $\langle \text{proof} \rangle$

lemma *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 $\text{null-space } A = \text{complex-matrix.orthogonal-complement } (\text{row-space } (\chi \ i \ j. \text{cnj } (A \ \$ \ i \ \$ \ j)))$
 $\langle \text{proof} \rangle$

lemma *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 $\text{left-null-space } A = \text{complex-matrix.orthogonal-complement } (\text{col-space } (\chi \ i \ j. \text{cnj } (A \ \$ \ i \ \$ \ j)))$
 $\langle \text{proof} \rangle$

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 $\text{null-space } A = \text{real-matrix.orthogonal-complement } (\text{row-space } A)$
 $\langle \text{proof} \rangle$

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 $\text{left-null-space } A = \text{real-matrix.orthogonal-complement } (\text{col-space } A)$
 $\langle \text{proof} \rangle$

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* $A\ k$
 $= (\chi\ a\ b.\ (if\ b = from\ nat\ k$
then $column\ b\ A - sum\ list\ (map\ (\lambda x.\ ((column\ b\ A \cdot x) / (x \cdot x)) *_{R}\ x)$
 $((map\ (\lambda n.\ column\ (from\ nat\ n)\ A)\ [0..<to\ nat\ b])))\ else\ column\ b\ A)\ \$\ a)$

15.1.2 General properties about *sum-list*

lemma *sum-list-remdups*:
assumes $!!i\ j.\ i < length\ xs \wedge j < length\ xs \wedge i \neq j$
 $\wedge xs\ !\ i = xs\ !\ j \longrightarrow xs\ !\ i = 0 \wedge xs\ !\ j = 0$
shows $sum\ list\ (remdups\ xs) = sum\ list\ xs$
 $\langle proof \rangle$

lemma *sum-list-remdups-2*:
fixes $f :: 'a :: \{zero, monoid\text{-}add\} \Rightarrow 'a$
assumes $!!i\ j.\ i < length\ xs \wedge j < length\ xs \wedge i \neq j \wedge (xs\ !\ i) = (xs\ !\ j)$
 $\longrightarrow f\ (xs\ !\ i) = 0 \wedge f\ (xs\ !\ j) = 0$
shows $sum\ list\ (map\ f\ (remdups\ xs)) = sum\ list\ (map\ f\ xs)$
 $\langle proof \rangle$

15.1.3 Proving a code equation to improve the performance

lemma *set-map-column*:
 $set\ (map\ (\lambda n.\ column\ (from\ nat\ n)\ G)\ [0..<to\ nat\ b]) = \{column\ i\ G \mid i. i < b\}$
 $\langle proof \rangle$

lemma *column-Gram-Schmidt-column-k-repeated-0*:
fixes $A :: 'a :: \{real\text{-}inner\} \wedge 'n :: \{mod\text{-}type\} \wedge 'm :: \{mod\text{-}type\}$
assumes $i\text{-not-}k: i \neq k$ **and** $ik: i < k$
and $c\text{-eq}: column\ k\ (Gram\ Schmidt\ column\ k\ A\ (to\ nat\ k))$
 $= column\ i\ (Gram\ Schmidt\ column\ k\ A\ (to\ nat\ k))$
and o : pairwise orthogonal $\{column\ i\ A \mid i. i < k\}$
shows $column\ k\ (Gram\ Schmidt\ column\ k\ A\ (to\ nat\ k)) = 0$

and $\text{column } i \text{ (Gram-Schmidt-column-} k \text{ } A \text{ (to-nat } k)) = 0$
 ⟨proof⟩

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** $j: j \leq \text{from-nat } k$
and $c\text{-eq}: \text{column } j \text{ (Gram-Schmidt-upt-} k \text{ } A \text{ } k)$
 $= \text{column } i \text{ (Gram-Schmidt-upt-} k \text{ } A \text{ } k)$
and $k: k < \text{ncols } A$
shows $\text{column } j \text{ (Gram-Schmidt-upt-} k \text{ } A \text{ } k) = 0$
 ⟨proof⟩

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** $j: j \leq k$
and $c\text{-eq}: \text{column } j \text{ (Gram-Schmidt-upt-} k \text{ } A \text{ (to-nat } k))$
 $= \text{column } i \text{ (Gram-Schmidt-upt-} k \text{ } A \text{ (to-nat } k))$
shows $\text{column } j \text{ (Gram-Schmidt-upt-} k \text{ } A \text{ (to-nat } k)) = 0$
 ⟨proof⟩

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 $c\text{-eq}: \text{column } j \text{ (Gram-Schmidt-upt-} k \text{ } A \text{ (to-nat } k))$
 $= \text{column } i \text{ (Gram-Schmidt-upt-} k \text{ } A \text{ (to-nat } k))$
shows $\text{column } j \text{ (Gram-Schmidt-upt-} k \text{ } A \text{ (to-nat } k)) = 0$
and $\text{column } i \text{ (Gram-Schmidt-upt-} k \text{ } A \text{ (to-nat } k)) = 0$
 ⟨proof⟩

lemma *column-Gram-Schmidt-column-k-eq-efficient*:
fixes $A::\text{real}^n::\{\text{mod-type}\}^m::\{\text{mod-type}\}$
assumes $\text{Gram-Schmidt-upt-} k \text{ } A \text{ } k = \text{foldl Gram-Schmidt-column-} k \text{-efficient } A$
 $[0..<\text{Suc } k]$
and $\text{suc-}k: \text{Suc } k < \text{ncols } A$
shows $\text{column } b \text{ (Gram-Schmidt-column-} k \text{ (Gram-Schmidt-upt-} k \text{ } A \text{ } k) \text{ (Suc } k))$
 $= \text{column } b \text{ (Gram-Schmidt-column-} k \text{-efficient (Gram-Schmidt-upt-} k \text{ } A \text{ } k) \text{ (Suc } k))$
 ⟨proof⟩

lemma *Gram-Schmidt-upt-k-efficient-induction*:
fixes $A::\text{real}^n::\{\text{mod-type}\}^m::\{\text{mod-type}\}$

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 *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)
<proof>

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]
<proof>

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 = foldl Gram-Schmidt-column-k-efficient A [0..<ncols*
A]
<proof>

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 =*
*tabulate2 (nrows-iarray A) (ncols-iarray A) ($\lambda a b$. let *column-b-A = column-iarray*
b A in
(if b = k then (column-b-A - sum-list (map (λx . ((column-b-A \cdot i x) / (x \cdot i x))
**_R x)*
((List.map (λn . column-iarray n A) [0..<b])))
*else column-b-A) !! a)**

definition *Gram-Schmidt-matrix-iarrays-efficient A*
= foldl Gram-Schmidt-column-k-iarrays-efficient A [0..<ncols-iarray A]

definition *QR-decomposition-iarrays-efficient A =*
(let Q = divide-by-norm-iarray (Gram-Schmidt-matrix-iarrays-efficient A)
*in (Q, transpose-iarray Q **i A))*

15.2.2 General properties

lemma *tabulate2-nth*:

assumes $i: i < nr$ and $j: j < nc$

shows $(\text{tabulate2 } nr \ nc \ f) !! i !! j = f \ i \ j$

<proof>

lemma *vec-to-iarray-minus*^[code-unfold]:

$\text{vec-to-iarray } (a - b) = (\text{vec-to-iarray } a) - (\text{vec-to-iarray } b)$

<proof>

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>

lemma *sum-list-map-vec-to-iarray*:

assumes $xs \neq []$

shows $\text{sum-list } (\text{map } (\text{vec-to-iarray} \circ f) \ xs) = \text{vec-to-iarray } (\text{sum-list } (\text{map } f \ xs))$

<proof>

15.2.3 Proving the equivalence

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

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

assumes $k: k < ncols \ A$

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

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

<proof>

lemma *matrix-to-iarray-Gram-Schmidt-upt-k-efficient*:

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

assumes $k: k < ncols \ A$

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

$= \text{foldl } \text{Gram-Schmidt-column-k-iarrays-efficient } (\text{matrix-to-iarray } A) \ [0..<Suc$

$k]$

<proof>

lemma *matrix-to-iarray-Gram-Schmidt-matrix-efficient*^[code-unfold]:

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

shows $\text{matrix-to-iarray } (\text{Gram-Schmidt-matrix } A)$

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

<proof>

lemma *QR-decomposition-iarrays-efficient*[code]:
QR-decomposition-iarrays (matrix-to-iarray A)
= *QR-decomposition-iarrays-efficient* (matrix-to-iarray A)
⟨proof⟩

15.3 Other code equations that improve the performance

lemma *inner-iarray-code*[code]:
inner-iarray A B = *sum-list* (map (λn. A !! n * B !! n) [0..*IArray.length* A])
⟨proof⟩

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 (λx. ((col-k · i x) / (x · i x)) *_R x)
((*List.map* (λn. *column-iarray* n A) [0..*k*]))))
in (λa 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
⟨proof⟩

end