QR Decomposition

By Jose Divasón and Jesús Aransay*

March 17, 2025

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

*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).

1 Miscellaneous file for the QR algorithm

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

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

lemma [*code abstract*]: *vec-nth* $(a - b) = (\%i.\ a\$i - b\$i)$ ⟨*proof*⟩
lemma [*code abstract*]: *vec-nth* $(c *_R x) = (\lambda i.\ c *_R (x\$i))$ ⟨*proof*⟩

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

lemma *from-nat-le*:
  fixes *i*::*'a*::{*mod-type*}
  assumes *i*: *to-nat i*< *k*
  and *k*: *k*<*CARD*(*'a*)
  shows *i* < *from-nat k*
  ⟨*proof*⟩

Some properties about orthogonal matrices.

lemma *orthogonal-mult*:
  assumes *orthogonal a b*
  shows *orthogonal* ($x *_R a$) ($y *_R b$)
  ⟨*proof*⟩

lemma *orthogonal-matrix-is-orthogonal*:
  fixes *A*::*real^'n^'n*
  assumes *o*: *orthogonal-matrix A*
  shows (*pairwise orthogonal* (*columns A*))
⟨*proof*⟩

lemma *orthogonal-matrix-norm*:
  fixes *A*::*real^'n^'n*
  assumes *o*: *orthogonal-matrix A*
  shows *norm* (*column i A*) = *1*
⟨*proof*⟩

lemma *orthogonal-matrix-card*:
  fixes *A*::*real^'n^'n*
  assumes *o*: *orthogonal-matrix A*
  shows *card* (*columns A*) = *ncols A*
⟨*proof*⟩

lemma *orthogonal-matrix-intro*:
  fixes *A*::*real^'n^'n*
  assumes *p*: (*pairwise orthogonal* (*columns A*))
  and *n*: ∀ *i*. *norm* (*column i A*) = *1*
  and *c*: *card* (*columns A*) = *ncols A*
  shows *orthogonal-matrix A*
⟨*proof*⟩

lemma *orthogonal-matrix2*:
  fixes *A*::*real^'n^'n*
  shows *orthogonal-matrix A* = ((*pairwise orthogonal* (*columns A*)) ∧ (∀ *i*. *norm* (*column i A*) = *1*) ∧
  (*card* (*columns A*) = *ncols A*))
  ⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

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

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

lemma *invertible-transpose-mult*:
  fixes $A::real^{\frown}'cols::\{mod\text{-}type\}^{\frown}'rows::\{mod\text{-}type\}$
  assumes r: $rank\ A = ncols\ A$
  shows *invertible* $(transpose\ A ** A)$
$\langle proof \rangle$

lemma *matrix-inv-mult*:
  fixes $A::'a::\{semiring\text{-}1\}^{\frown}'n^{\frown}'n$
  and $B::'a::\{semiring\text{-}1\}^{\frown}'n^{\frown}'n$
  assumes *invertible* A **and** *invertible* B
  shows $matrix\text{-}inv\ (A ** B) = matrix\text{-}inv\ B ** matrix\text{-}inv\ A$
$\langle proof \rangle$

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

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*: $(span\ S = span\ T) = (S \subseteq span\ T \wedge T \subseteq span\ S)$
  $\langle proof \rangle$

end

lemma *basis-orthogonal*:
  fixes B :: $'a$::*real-inner set*
  assumes *fB*: *finite B*
  shows $\exists\, C.$ *finite C* $\wedge$ *card C* $\leq$ *card B* $\wedge$ *span C*
    $=$ *span B* $\wedge$ *pairwise orthogonal C*
  (**is** $\exists\, C.$ *?P B C*)
  ⟨*proof*⟩

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

end

2 Projections

theory *Projections*
imports
  Miscellaneous-QR
begin

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

definition *proj v u* $=$ $(v \cdot u \;/\; (u \cdot u)) *_R u$

definition *proj-onto a S* $=$ (*sum* ($\lambda x.$ *proj a x*) *S*)

2.2 Properties

lemma *proj-onto-sum-rw*:
  sum ($\lambda x.$ $(x \cdot v \;/\; (x \cdot x)) *_R x$) *A* $=$ *sum* ($\lambda x.$ $(v \cdot x \;/\; (x \cdot x)) *_R x$) *A*
  ⟨*proof*⟩

lemma *vector-sub-project-orthogonal-proj*:
  fixes $b\ x$ :: $'a$::*euclidean-space*
  shows *inner b* $(x -$ *proj x b*$) = 0$
  ⟨*proof*⟩

lemma *orthogonal-proj-set*:
  assumes *yC*: $y \in C$ **and** *C*: *finite C* **and** *p*: *pairwise orthogonal C*
  shows *orthogonal* $(a -$ *proj-onto a C*$)$ *y*
⟨*proof*⟩

lemma *pairwise-orthogonal-proj-set*:
  assumes *C*: *finite C* **and** *p*: *pairwise orthogonal C*
  shows *pairwise orthogonal* (*insert* $(a -$ *proj-onto a C*$)$ *C*)

⟨*proof*⟩

2.3 Orthogonal Complement

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

lemma *in-orthogonal-complement-imp-orthogonal*:
assumes x: $y \in S$
and $x \in$ *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* $B \subseteq$ *orthogonal-complement* A
⟨*proof*⟩

lemma *B-in-orthogonal-complement-of-orthogonal-complement*:
shows $B \subseteq$ *orthogonal-complement* (*orthogonal-complement* B)
⟨*proof*⟩

lemma *phytagorean-theorem-norm*:
assumes o: *orthogonal* x y
shows *norm* $(x+y)\hat{}2$=*norm* $x\hat{}2$ + *norm* $y\hat{}2$
⟨*proof*⟩

lemma *in-orthogonal-complement-basis*:
fixes B::'a::{*euclidean-space*} *set*
assumes S: *subspace* S
and *ind-B*: *independent* B
and B: $B \subseteq S$
and *span-B*: $S \subseteq$ *span* B
shows ($v \in$ *orthogonal-complement* S) = ($\forall\, a{\in}B.$ *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::{*euclidean-space*} *set*
assumes *subspace-S*: *subspace* S
and *ind-X*: *independent* X
and X: $X \subseteq S$
and *span-X*: $S \subseteq$ *span* X

and *o*: *pairwise orthogonal X*
shows $(v - \textit{proj-onto } v\ X) \in \textit{orthogonal-complement } S$
⟨*proof*⟩

Part 2 of the Theorem 1.7 in the previous website.

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

2.4 Normalization of vectors

definition *normalize*
where $\textit{normalize } x = ((1/\textit{norm } x) *_R x)$
definition *normalize-set-of-vec*
where *normalize-set-of-vec X* = *normalize‘ X*

lemma *norm-normalize*:
assumes $x \neq 0$
shows $\textit{norm } (\textit{normalize } x) = 1$
⟨*proof*⟩

lemma *normalize-0*: $(\textit{normalize } x = 0) = (x = 0)$
⟨*proof*⟩

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

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 fist 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* :: $('a::\{real\text{-}inner\}^{\frown}'b)$ => $('a^{\frown}'b)$ *list* => $('a^{\frown}'b)$ *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*::$('a::\{real\text{-}inner\}^{\frown}'b)$ *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*::$('a::\{real\text{-}inner\}^{\frown}'b)$ *list*
  assumes *distinct xs*
  shows *card*(*set* (*Gram-Schmidt xs*)) ≤ *card* (*set* (*xs*))
  ⟨*proof*⟩

lemma *orthogonal-basis-exists*:
  fixes *V* :: $(real^{\frown}'b)$ *list*
  assumes *B*: *is-basis* (*set V*)

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

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

3.1.2 Second way

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

definition *Gram-Schmidt2 xs* $=$ *Gram-Schmidt* (*rev xs*)

lemma *basis-orthogonal2*:
fixes *xs*::('*a*::{*real-inner*}$^{\frown\prime}$*b*) *list*
shows *length* (*Gram-Schmidt2 xs*) $=$ *length* (*xs*)
$\wedge$ *span* (*set* (*Gram-Schmidt2 xs*)) $=$ *span* (*set xs*)
$\wedge$ *pairwise orthogonal* (*set* (*Gram-Schmidt2 xs*))
$\langle proof \rangle$

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

lemma *orthogonal-basis-exists2*:
fixes V :: (*real*$^{\frown\prime}$*b*) *list*
assumes B: *is-basis* (*set* V)
and d: *distinct* V
shows *vec.independent* (*set* (*Gram-Schmidt2* V)) $\wedge$ (*set* V) $\subseteq$ *vec.span* (*set* (*Gram-Schmidt2* V))
$\wedge$ (*card* (*set* (*Gram-Schmidt2* V)) $=$ *vec.dim* (*set* V)) $\wedge$ *pairwise orthogonal* (*set* (*Gram-Schmidt2* V))
$\langle proof \rangle$

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* :: $'a::\{real\text{-}inner\}^{\frown}'cols::\{mod\text{-}type\}^{\frown}'rows \Rightarrow$ *nat*
$\Rightarrow$ $'a^{\frown}'cols::\{mod\text{-}type\}^{\frown}'rows$
where *Gram-Schmidt-column-k A k*
$= (\chi\ a.\ (\chi\ b.\ ($*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* $-$ ($\sum x \in \{$*column i A*$|i.\ i < b\}$. ((*column b A*) $\cdot$ *x* / (*x* $\cdot$ *x*)) $*_R$ *x*))
else (*column b A*)) \$ *a*)

lemma *Gram-Schmidt-column-k-row-code*[*code abstract*]:
vec-nth (*Gram-Schmidt-column-k-row A k a*)
= (%*b*. (*if* $b =$ *from-nat k*
then (*column b A* $-$ ($\sum x \in \{$*column i A*$|i.\ i < b\}$. ((*column b A*) $\cdot$ *x* / (*x* $\cdot$ *x*)) $*_R$ *x*))
else (*column b A*)) \$ *a*)
⟨*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 $A::'a::\{real\text{-}inner\}^{\frown}'cols::\{mod\text{-}type\}^{\frown}'rows$
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 $A::'a::\{real\text{-}inner\}^{\frown}'cols::\{mod\text{-}type\}^{\frown}'rows$
assumes k: $Suc\ k < CARD\ ('cols)$
shows $\{column\ i\ (Gram\text{-}Schmidt\text{-}upt\text{-}k\ A\ (Suc\ k))\ |i.\ to\text{-}nat\ i \leq (Suc\ k)\} =$
$\{column\ i\ (Gram\text{-}Schmidt\text{-}upt\text{-}k\ A\ k)\ |i.\ to\text{-}nat\ i \leq k\} \cup \{column\ (from\text{-}nat\ (Suc\ k))\ (Gram\text{-}Schmidt\text{-}upt\text{-}k\ A\ k)$
$- (\sum x \in \{column\ i\ (Gram\text{-}Schmidt\text{-}upt\text{-}k\ A\ k)\ |i.\ to\text{-}nat\ i \leq k\}.\ (x \cdot (column\ (from\text{-}nat\ (Suc\ k))\ (Gram\text{-}Schmidt\text{-}upt\text{-}k\ A\ k))\ /\ (x \cdot x))\ *_R\ x)\}$
⟨*proof*⟩

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

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

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

corollary *orthogonal-Gram-Schmidt-matrix2*:
shows *pairwise orthogonal* (*columns* (*Gram-Schmidt-matrix A*))
⟨*proof*⟩

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

lemma *column-Gram-Schmidt-column-k'*:
fixes $A::'a::\{real\text{-}inner\}^{\frown}'n::\{mod\text{-}type\}^{\frown}'m::\{mod\text{-}type\}$
assumes *i-not-k*: $i \neq k$
shows $column\ i\ (Gram\text{-}Schmidt\text{-}column\text{-}k\ A\ (to\text{-}nat\ k)) = (column\ i\ A)$
⟨*proof*⟩

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

lemma *cols-upt-k-insert*:

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

lemma *columns-eq-cols-upt-k*:
fixes A::$'a$^$'cols$::{*mod-type*}^$'rows$::{*mod-type*}
shows *cols-upt-k A* (*ncols A* − *1*) = *columns A*
⟨*proof*⟩

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

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

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

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

lemma *is-basis-columns-Gram-Schmidt-matrix*:
fixes A::*real*^$'n$::{*mod-type*}^$'m$::{*mod-type*}
assumes b: *is-basis* (*columns A*)
and c: *card* (*columns A*) = *ncols A*
shows *is-basis* (*columns* (*Gram-Schmidt-matrix A*))
∧ *card* (*columns* (*Gram-Schmidt-matrix A*)) = *ncols A*
⟨*proof*⟩

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 $A::real^{\frown}'n::\{mod\text{-}type\}^{\frown}'m::\{mod\text{-}type\}$
assumes $i>k$
and i: $i<ncols\ A$
shows $column\ (from\text{-}nat\ i)\ (Gram\text{-}Schmidt\text{-}upt\text{-}k\ A\ k) = column\ (from\text{-}nat\ i)\ A$
⟨*proof*⟩

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

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

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

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

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

⟨*proof*⟩

corollary *column-Gram-Schmidt-matrix2*:
fixes *A*::*real*^'*n*::{*mod-type*}^'*m*::{*mod-type*}
shows (*column k A*) = *column k* (*Gram-Schmidt-matrix A*)
+ ($\sum$ *x*∈{*column i* (*Gram-Schmidt-matrix A*)|*i*. *i* < *k*}. (*x* · (*column k A*) / (*x* · *x*)) $*_R$ *x*)
⟨*proof*⟩

lemma *independent-columns-Gram-Schmidt-matrix*:
fixes *A*::*real*^'*n*::{*mod-type*}^'*m*::{*mod-type*}
assumes *b*: *vec.independent* (*columns A*)
and *c*: *card* (*columns A*) = *ncols A*
shows *vec.independent* (*columns* (*Gram-Schmidt-matrix A*)) ∧ *card* (*columns* (*Gram-Schmidt-matrix A*)) = *ncols A*
⟨*proof*⟩

lemma *column-eq-Gram-Schmidt-matrix*:
fixes *A*::*real*^'*n*::{*mod-type*}^'*m*::{*mod-type*}
assumes *r*: *rank A* = *ncols A*
and *c*: *column i* (*Gram-Schmidt-matrix A*) = *column ia* (*Gram-Schmidt-matrix A*)
shows *i* = *ia*
⟨*proof*⟩

lemma *scaleR-columns-Gram-Schmidt-matrix*:
fixes *A*::*real*^'*n*::{*mod-type*}^'*m*::{*mod-type*}
assumes *i* ≠ *j*
and *rank A* = *ncols A*
shows *column j* (*Gram-Schmidt-matrix A*) · *column i* (*Gram-Schmidt-matrix A*) = *0*
⟨*proof*⟩

3.1.4 Examples of execution

Code lemma

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

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

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

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

end

4 QR Decomposition

theory *QR-Decomposition*
imports *Gram-Schmidt*
begin

4.1 The QR Decomposition of a matrix

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

Although in some texts the QR decomposition is presented over square matrices, it can be applied to any matrix. There are some variants of the algorithm, depending on the properties that the output matrices satisfy (see for instance, http://inst.eecs.berkeley.edu/~ee127a/book/login/l_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 demostrate 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* = (χ *a b. normalize* (*column b A*) \$ *a*)

Properties

lemma *norm-column-divide-by-norm*:
 fixes A::'a::{*real-inner*}^'*cols*^'*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 an 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*)))
∧ *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::$real$^$'n$::{$mod\text{-}type$}^$'m$::{$mod\text{-}type$}
shows $col\text{-}space\ A = col\text{-}space\ (fst\ (QR\text{-}decomposition\ A))$
⟨$proof$⟩

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

lemma $orthogonal\text{-}matrix\text{-}fst\text{-}QR\text{-}decomposition$:
fixes A::$real$^$'n$::{$mod\text{-}type$}^$'m$::{$mod\text{-}type$}
assumes r: $rank\ A = ncols\ A$
shows $transpose\ (fst\ (QR\text{-}decomposition\ A))\ **\ (fst\ (QR\text{-}decomposition\ A)) = mat\ 1$
⟨$proof$⟩

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

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

corollary $column\text{-}QR\text{-}decomposition$:
fixes A::$real$^$'n$::{$mod\text{-}type$}^$'m$::{$mod\text{-}type$}
assumes r: $rank\ A = ncols\ A$
shows $column\ k\ ((Gram\text{-}Schmidt\text{-}matrix\ A))$
$= (column\ k\ A) - (\sum x{\in}\{column\ i\ (fst\ (QR\text{-}decomposition\ A))|i.\ i < k\}.\ (x \cdot (column\ k\ A)\ /\ (x \cdot x))\ *_R\ x)$
⟨$proof$⟩

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

$\cdot$ *x*)) $*_R$ *x*)
⟨*proof*⟩

lemma *norm-uk-eq*:
fixes *A*::*real*^'*n*::{*mod-type*}^'*m*::{*mod-type*}
assumes *r*: *rank A* = *ncols A*
shows *norm* (*column k* ((*Gram-Schmidt-matrix A*))) = ((*column k* (*fst*(*QR-decomposition A*))) $\cdot$ (*column k A*))
⟨*proof*⟩

corollary *column-QR-decomposition2*:
fixes *A*::*real*^'*n*::{*mod-type*}^'*m*::{*mod-type*}
assumes *r*: *rank A* = *ncols A*
shows (*column k A*)
= ($\sum$ *x*∈{*column i* (*fst* (*QR-decomposition A*))|*i*. *i* ≤ *k*}. (*x* $\cdot$ (*column k A*)) $*_R$ *x*)
⟨*proof*⟩

lemma *orthogonal-columns-fst-QR-decomposition*:
assumes *i-not-ia*: (*column i* (*fst* (*QR-decomposition A*))) ≠ (*column ia* (*fst* (*QR-decomposition A*)))
shows (*column i* (*fst* (*QR-decomposition A*)) $\cdot$ *column ia* (*fst* (*QR-decomposition A*))) = *0*
⟨*proof*⟩

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

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

lemma *sums-columns-Q-0*:
fixes *A*::*real*^'*n*::{*mod-type*}^'*m*::{*mod-type*}
assumes *r*: *rank A* = *ncols A*
shows ($\sum$ *x*∈{*column i* (*fst* (*QR-decomposition A*)) |*i*. *i*>*b*}. *x* $\cdot$ *column b A* * *x* $ *a*) = *0*
⟨*proof*⟩

lemma *QR-decomposition-mult*:
fixes *A*::*real*^'*n*::{*mod-type*}^'*m*::{*mod-type*}

assumes *r*: *rank A* = *ncols A*
shows *A* = (*fst* (*QR-decomposition A*)) ∗∗ (*snd* (*QR-decomposition A*))
⟨*proof*⟩

lemma *upper-triangular-snd-QR-decomposition*:
fixes *A*::*real*^'*n*::{*mod-type*}^'*m*::{*mod-type*}
assumes *r*: *rank A* = *ncols A*
shows *upper-triangular* (*snd* (*QR-decomposition A*))
⟨*proof*⟩

lemma *upper-triangular-invertible*:
fixes *A* :: *real*^'*n*::{*finite,wellorder*}^'*n*::{*finite,wellorder*}
assumes *u*: *upper-triangular A*
and *d*: ∀ *i*. *A* $ *i* $ *i* ≠ *0*
shows *invertible A*
⟨*proof*⟩

lemma *invertible-snd-QR-decomposition*:
fixes *A*::*real*^'*n*::{*mod-type*}^'*m*::{*mod-type*}
assumes *r*: *rank A* = *ncols A*
shows *invertible* (*snd* (*QR-decomposition A*))
⟨*proof*⟩

lemma *QR-decomposition*:
fixes *A*::*real*^'*n*::{*mod-type*}^'*m*::{*mod-type*}
assumes *r*: *rank A* = *ncols A*
shows *A* = *fst* (*QR-decomposition A*) ∗∗ *snd* (*QR-decomposition A*) ∧
pairwise orthogonal (*columns* (*fst* (*QR-decomposition A*))) ∧
(∀ *i*. *norm* (*column i* (*fst* (*QR-decomposition A*))) = *1*) ∧
(*transpose* (*fst* (*QR-decomposition A*))) ∗∗ (*fst* (*QR-decomposition A*)) = *mat 1*
∧
vec.independent (*columns* (*fst* (*QR-decomposition A*))) ∧
col-space A = *col-space* (*fst* (*QR-decomposition A*)) ∧
card (*columns A*) = *card* (*columns* (*fst* (*QR-decomposition A*))) ∧
invertible (*snd* (*QR-decomposition A*)) ∧
upper-triangular (*snd* (*QR-decomposition A*))
⟨*proof*⟩

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

pairwise orthogonal (*columns* (*fst* (*QR-decomposition* A))) $\wedge$
($\forall$ *i*. *norm* (*column i* (*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*::*real*^'*n*::{*mod-type*}^'*n*::{*mod-type*}
assumes *r*: *rank A* = *ncols A*
shows |*det A*| = |(*prod* (λ*i*. *snd*(*QR-decomposition A*)\$*i*\$*i*) (*UNIV*::'*n set*))|
⟨*proof*⟩

end

5 Least Squares Approximation

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

5.1 Second part of the Fundamental Theorem of Linear Algebra

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

lemma *null-space-orthogonal-complement-row-space*:
fixes *A*::*real*^'*cols*^'*rows*::{*finite*,*wellorder*}
shows *null-space A* = *orthogonal-complement* (*row-space A*)
⟨*proof*⟩

lemma *left-null-space-orthogonal-complement-col-space*:
fixes *A*::*real*^'*cols*::{*finite*,*wellorder*}^'*rows*
shows *left-null-space A* = *orthogonal-complement* (*col-space A*)
⟨*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*::{*euclidean-space*} *set*
assumes *subspace-S*: *subspace S*
and *ind-X*: *independent X*
and *X*: *X* ⊆ *S*

and *span-X*: $S \subseteq$ *span X*
and *o*: *pairwise orthogonal X*
and *not-eq*: *proj-onto v X* $\neq$ *y*
and *y*: $y \in S$
shows *norm* ($v -$ *proj-onto v X*) $<$ *norm* ($v - y$)
⟨*proof*⟩

lemma *least-squares-approximation2*:
fixes *S*::′*a*::{*euclidean-space*} *set*
assumes *subspace-S*: *subspace S*
and *y*: $y \in S$
shows $\exists p \in S$. *norm* ($v - p$) $\leq$ *norm* ($v - y$) $\wedge$ ($v-p$) $\in$ *orthogonal-complement S*
⟨*proof*⟩

corollary *least-squares-approximation3*:
fixes *S*::′*a*::{*euclidean-space*} *set*
assumes *subspace-S*: *subspace S*
shows $\exists p \in S$. $\forall y \in S$. *norm* ($v - p$) $\leq$ *norm* ($v - y$) $\wedge$ ($v-p$) $\in$ *orthogonal-complement S*
⟨*proof*⟩

lemma *norm-least-squares*:
fixes *A*::*real*^′*cols*::{*finite*,*wellorder*}^′*rows*
shows $\exists x$. $\forall x'$. *norm* ($b - A *v\ x$) $\leq$ *norm* ($b - A *v\ x'$)
⟨*proof*⟩

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

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

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

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

⟨*proof*⟩

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

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

corollary *least-squares-approximation7*:
fixes S::'a::{*euclidean-space*} *set*
assumes *subspace-S*: *subspace S*
and $v - p \in$ *orthogonal-complement S*
and $p\in S$
and $y \in S$
shows *norm* $(v - p) \leq$ *norm* $(v - y)$
⟨*proof*⟩

lemma *in-set-least-squares-approximation*:
fixes A::*real*^'*cols*::{*finite*, *wellorder*}^'*rows*
assumes o: $A *v\ x - b \in$ *orthogonal-complement* (*col-space* A)
shows ($x \in$ *set-least-squares-approximation* A b)
⟨*proof*⟩

lemma *in-set-least-squares-approximation-eq*:
fixes A::*real*^'*cols*::{*finite*,*wellorder*}^'*rows*
shows ($x \in$ *set-least-squares-approximation* A b) = (*transpose* A ** A *v x = *transpose* A *v b)
⟨*proof*⟩

lemma *in-set-least-squares-approximation-eq-full-rank*:
fixes A::*real*^'*cols*::*mod-type*^'*rows*::*mod-type*
assumes r: *rank* A = *ncols* A
shows ($x \in$ *set-least-squares-approximation* A b) = (x = *matrix-inv* (*transpose* A ** A)***transpose* A *v b)
⟨*proof*⟩

lemma *in-set-least-squares-approximation-eq-full-rank-QR*:
  fixes *A*::*real*^'*cols*::{*mod-type*}^'*rows*::{*mod-type*}
  assumes *r*: *rank A* = *ncols A*
  shows (*x* ∈ *set-least-squares-approximation A b*) = ((*snd* (*QR-decomposition A*)) ∗*v x* = *transpose* (*fst* (*QR-decomposition A*)) ∗*v b*)
⟨*proof*⟩

corollary *in-set-least-squares-approximation-eq-full-rank-QR2*:
  fixes *A*::*real*^'*cols*::{*mod-type*}^'*rows*::{*mod-type*}
  assumes *r*: *rank A* = *ncols A*
  shows (*x* ∈ *set-least-squares-approximation A b*) = (*x* = *matrix-inv* (*snd* (*QR-decomposition A*)) ∗∗ *transpose* (*fst* (*QR-decomposition A*)) ∗*v b*)
⟨*proof*⟩

lemma *set-least-squares-approximation-unique-solution*:
  fixes *A*::*real*^'*cols*::{*mod-type*}^'*rows*::{*mod-type*}
  assumes *r*: *rank A* = *ncols A*
  shows (*set-least-squares-approximation A b*) = {*matrix-inv* (*transpose A* ∗∗ *A*)∗∗*transpose A* ∗*v b*}
  ⟨*proof*⟩

lemma *set-least-squares-approximation-unique-solution-QR*:
  fixes *A*::*real*^'*cols*::{*mod-type*}^'*rows*::{*mod-type*}
  assumes *r*: *rank A* = *ncols A*
  shows (*set-least-squares-approximation A b*) = {*matrix-inv* (*snd* (*QR-decomposition A*)) ∗∗ *transpose* (*fst* (*QR-decomposition A*)) ∗*v b*}
  ⟨*proof*⟩

end

6 Examples of execution using floats

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

6.0.1 Examples

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

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

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

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

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

export-code *example1 example2 example3 example4 example5* **in** *SML* **module-name** *QR*

end

7 Examples of execution using symbolic computation

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

7.1 Execution of the QR decomposition using symbolic computation

7.1.1 Some previous definitions and lemmas

The symbolic computation is based on the René Thiemann's work about implementing field extensions of the form Q[sqrt(b)].

definition *show-vec-real v* = (χ *i*. *show-real* (*v* $ *i*))

lemma [*code abstract*]: *vec-nth* (*show-vec-real v*) = (% *i*. *show-real* (*v* $ *i*))
⟨*proof*⟩

definition *show-matrix-real A* = (χ *i*. *show-vec-real* (*A* $ *i*))

lemma[*code abstract*]: *vec-nth* (*show-matrix-real A*) = (% *i*. *show-vec-real* (*A* $ *i*))
⟨*proof*⟩

7.1.2 Examples

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

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

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

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

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

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

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

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

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

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

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

export-code *example1* **in** *SML* **module-name** *QR*

end

8 IArray Addenda QR

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

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

In order to make the formalisation of the QR algorithm easier, we have decided to present here some alternative instantiatons 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* (λ*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* (λ*a*. *if a* < *length-A then A*!!*a else 0*) *n*;
B′=*IArray.of-fun* (λ*a*. *if a* < *length-B then B*!!*a else 0*) *n*
in IArray.of-fun (λ*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* (λ*a*. *if a* < *length-A then A*!!*a else 0*) *n*;
  B'=*IArray.of-fun* (λ*a*. *if a* < *length-B then B*!!*a else 0*) *n*
  in
  IArray.of-fun (λ*a*. *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* (λ*n*. − *A*!!*n*) (*IArray.length A*)
instance ⟨*proof*⟩
end

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

definition *minus-iarray* :: *'a iarray* ⇒ *'a iarray* ⇒ *'a iarray*
  where *minus-iarray A B* =

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

instance ⟨*proof*⟩
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* ⇀ (*SML*) *Vector.exists*
| **constant** *IArray-Addenda-QR.all* ⇀ (*SML*) *Vector.all*

end

9 Matrices as nested IArrays

theory *Matrix-To-IArray-QR*
imports
Rank-Nullity-Theorem.Mod-Type
Gauss-Jordan.Elementary-Operations
IArray-Addenda-QR
begin

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

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

9.1.1 Isomorphism between vec and iarray

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

lemma *vec-to-iarray-nth*:
  fixes A::$'a^{\frown}'n$::{*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::$'a^{\frown}'n$::{*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::$'a^{\frown}'n$::{*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::$'a^{\frown}'n$::{*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::$'a^{\frown}'n$::{*mod-type*}) = A
⟨*proof*⟩

lemma *length-vec-to-iarray*:
  fixes *xa*::$'a^{\frown}'n$::{*mod-type*}

shows *IArray.length* (*vec-to-iarray xa*) = *CARD*('*n*)
⟨*proof*⟩

9.1.2 Isomorphism between matrix and nested iarrays

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

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

lemma *matrix-to-iarray-morph*:
fixes *A*::'*a*^'*n*::{*mod-type*}^'*m*::{*mod-type*}
shows (*A* = *B*) = (*matrix-to-iarray A* = *matrix-to-iarray B*)
⟨*proof*⟩

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

lemma *map-vec-to-iarray-rw*[*simp*]:
fixes *A*::'*a*^'*columns*::{*mod-type*}^'*rows*::{*mod-type*}
shows *map* (λ*x*. *vec-to-iarray* (*A* $ *from-nat x*)) [*0*..<*CARD*('*rows*)] ! *to-nat i* = *vec-to-iarray* (*A* $ *i*)
⟨*proof*⟩

lemma *matrix-to-iarray-nth*:
matrix-to-iarray A !! *to-nat i* !! *to-nat j* = *A* $ *i* $ *j*
⟨*proof*⟩

lemma *vec-matrix*: *vec-to-iarray* (*A*$*i*) = (*matrix-to-iarray A*) !! (*to-nat i*)
⟨*proof*⟩

lemma *iarray-to-matrix-matrix-to-iarray*:
fixes *A*::'*a*^'*columns*::{*mod-type*}^'*rows*::{*mod-type*}
shows *iarray-to-matrix* (*matrix-to-iarray A*) = *A*
⟨*proof*⟩

9.2 Definition of operations over matrices implemented by iarrays

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

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

definition *column-iarray* :: *nat* => *'a iarray iarray* => *'a iarray*
  where *column-iarray k A = IArray.of-fun (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** ‹**i› *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-iarray A q* !! $a = q * A!!a$
⟨proof⟩

lemma *length-eq-card-rows*:
fixes $A::'a^{\frown}'columns::\{mod\text{-}type\}^{\frown}'rows::\{mod\text{-}type\}$
shows *IArray.length* (*matrix-to-iarray A*) = *CARD*('*rows*)
⟨proof⟩

lemma *nrows-eq-card-rows*:
fixes $A::'a^{\frown}'columns::\{mod\text{-}type\}^{\frown}'rows::\{mod\text{-}type\}$
shows *nrows-iarray* (*matrix-to-iarray A*) = *CARD*('*rows*)
⟨proof⟩

lemma *length-eq-card-columns*:
fixes $A::'a^{\frown}'columns::\{mod\text{-}type\}^{\frown}'rows::\{mod\text{-}type\}$
shows *IArray.length* (*matrix-to-iarray A* !! *0*) = *CARD* ('*columns*)
⟨proof⟩

lemma *ncols-eq-card-columns*:
fixes $A::'a^{\frown}'columns::\{mod\text{-}type\}^{\frown}'rows::\{mod\text{-}type\}$
shows *ncols-iarray* (*matrix-to-iarray A*) = *CARD*('*columns*)
⟨proof⟩

lemma *matrix-to-iarray-nrows*:
fixes $A::'a^{\frown}'columns::\{mod\text{-}type\}^{\frown}'rows::\{mod\text{-}type\}$
shows *nrows A* = *nrows-iarray* (*matrix-to-iarray A*)
⟨proof⟩

lemma *matrix-to-iarray-ncols*:
fixes $A::'a^{\frown}'columns::\{mod\text{-}type\}^{\frown}'rows::\{mod\text{-}type\}$
shows *ncols A* = *ncols-iarray* (*matrix-to-iarray A*)
⟨proof⟩

lemma *vec-to-iarray-row*[*code-unfold*]: *vec-to-iarray* (*row i A*) = *row-iarray* (*to-nat i*) (*matrix-to-iarray A*)
⟨proof⟩

lemma *vec-to-iarray-row'*: *vec-to-iarray* (*row i A*) = (*matrix-to-iarray A*) !! (*to-nat i*)
⟨proof⟩

lemma *vec-to-iarray-column*[*code-unfold*]: *vec-to-iarray* (*column i A*) = *column-iarray* (*to-nat i*) (*matrix-to-iarray A*)
⟨proof⟩

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

assumes *k*: *k*<*ncols A*
shows (*vec-to-iarray* (*column* (*from-nat k*) *A*)) = (*column-iarray k* (*matrix-to-iarray A*))
⟨*proof*⟩

lemma *column-iarray-nth*:
assumes *i*: *i*<*nrows-iarray A*
shows *column-iarray j A* !! *i* = *A* !! *i* !! *j*
⟨*proof*⟩

lemma *vec-to-iarray-rows*: *vec-to-iarray*‘ (*rows A*) = *rows-iarray* (*matrix-to-iarray A*)
⟨*proof*⟩

lemma *vec-to-iarray-columns*: *vec-to-iarray*‘ (*columns A*) = *columns-iarray* (*matrix-to-iarray A*)
⟨*proof*⟩

9.3 Definition of elementary operations

definition *interchange-rows-iarray* :: *'a iarray iarray* => *nat* => *nat* => *'a iarray iarray*
where *interchange-rows-iarray A a b* = *IArray.of-fun* (λ*n*. *if n*=*a then A*!!*b else if n*=*b then A*!!*a else A*!!*n*) (*IArray.length A*)

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

definition *row-add-iarray* :: *'a*::{*plus*, *times*,*zero*} *iarray iarray* => *nat* => *nat* => *'a* => *'a iarray iarray*
where *row-add-iarray A a b q* = *IArray.of-fun* (λ*n*. *if n*=*a then A*!!*a* + *mult-iarray* (*A*!!*b*) *q else A*!!*n*) (*IArray.length A*)

definition *interchange-columns-iarray* :: *'a iarray iarray* => *nat* => *nat* => *'a iarray iarray*
where *interchange-columns-iarray A a b* = *tabulate2* (*nrows-iarray A*) (*ncols-iarray A*) (λ*i j*. *if j* = *a then A* !! *i* !! *b else if j* = *b then A* !! *i* !! *a else A* !! *i* !! *j*)

definition *mult-column-iarray* :: *'a*::{*times*} *iarray iarray* => *nat* => *'a* => *'a iarray iarray*
where *mult-column-iarray A n q* = *tabulate2* (*nrows-iarray A*) (*ncols-iarray A*) (λ*i j*. *if j* = *n then A* !! *i* !! *j* ∗ *q else A* !! *i* !! *j*)

definition *column-add-iarray* :: *'a*::{*plus*, *times*} *iarray iarray* => *nat* => *nat* => *'a* => *'a iarray iarray*
where *column-add-iarray A n m q* = *tabulate2* (*nrows-iarray A*) (*ncols-iarray A*) (λ*i j*. *if j* = *n then A* !! *i* !! *n* + *A* !! *i* !! *m* ∗ *q else A* !! *i* !! *j*)

9.3.1 Code generator

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

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

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

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

lemma *matrix-to-iarray-matrix-matrix-mult*[*code-unfold*]:
fixes *A*::'*a*::{*semiring-1*}^'*m*::{*mod-type*}^'*n*::{*mod-type*} **and** *B*::'*a*^'*b*::{*mod-type*}^'*m*::{*mod-type*}
shows *matrix-to-iarray* (*A* ** *B*) = (*matrix-to-iarray A*) **i (*matrix-to-iarray B*)
⟨*proof*⟩

lemma *vec-to-iarray-matrix-matrix-mult*[*code-unfold*]:
fixes *A*::'*a*::{*semiring-1*}^'*m*::{*mod-type*}^'*n*::{*mod-type*} **and** *x*::'*a*^'*m*::{*mod-type*}
shows *vec-to-iarray* (*A* *v *x*) = (*matrix-to-iarray A*) *iv (*vec-to-iarray x*)
⟨*proof*⟩

lemma *vec-to-iarray-vector-matrix-mult*[*code-unfold*]:
fixes *A*::'*a*::{*semiring-1*}^'*m*::{*mod-type*}^'*n*::{*mod-type*} **and** *x*::'*a*^'*n*::{*mod-type*}
shows *vec-to-iarray* (*x* v* *A*) = (*vec-to-iarray x*) v*i (*matrix-to-iarray A*)
⟨*proof*⟩

lemma *matrix-to-iarray-interchange-rows*[*code-unfold*]:
fixes *A*::'*a*::{*semiring-1*}^'*columns*::{*mod-type*}^'*rows*::{*mod-type*}
shows *matrix-to-iarray* (*interchange-rows A i j*) = *interchange-rows-iarray* (*matrix-to-iarray A*) (*to-nat i*) (*to-nat j*)
⟨*proof*⟩

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

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

lemma *matrix-to-iarray-interchange-columns*[*code-unfold*]:
  fixes *A*::*'a*::{*semiring-1*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*}
  shows *matrix-to-iarray* (*interchange-columns A i j*) = *interchange-columns-iarray* (*matrix-to-iarray A*) (*to-nat i*) (*to-nat j*)
  ⟨*proof*⟩

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

lemma *matrix-to-iarray-column-add*[*code-unfold*]:
  fixes *A*::*'a*::{*semiring-1*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*}
  shows *matrix-to-iarray* (*column-add A i j q*) = *column-add-iarray* (*matrix-to-iarray A*) (*to-nat i*) (*to-nat j*) *q*
  ⟨*proof*⟩

end

10 Gram Schmidt over IArrays

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

10.1 Some previous definitions, lemmas and instantiations about iarrays

definition *iarray-of-iarray-to-list-of-list* :: *'a iarray iarray* => *'a list list*
  where *iarray-of-iarray-to-list-of-list A* = *map IArray.list-of* (*map* ((!!) *A*) [*0*..<*IArray.length A*])

instantiation *iarray* :: (*scaleR*) *scaleR*
begin
definition *scaleR-iarray k A* = *IArray.of-fun* ($\lambda i.\ k *_R$ (*A* !! *i*)) (*IArray.length A*)
instance ⟨*proof*⟩
end

instantiation *iarray* :: (*times*) *times*
begin
definition *times-iarray A B* = *IArray.of-fun* (λ*i*. *A*!!*i* $*$ *B* !! *i*) (*IArray.length A*)
instance ⟨*proof*⟩
end

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

lemma *minus-iarray-component*:
  assumes *iA*: *i*<*IArray.length A*
  and *iB*: *i*<*IArray.length B*
  shows (*A*−*B*) !! *i* = *A*!!*i* − *B*!!*i*
⟨*proof*⟩

lemma *length-plus-iarray*:
  IArray.length (*A*+*B*)=*max* (*IArray.length A*) (*IArray.length B*)
  ⟨*proof*⟩

lemma *length-sum-iarray*:
  assumes *finite S* **and** *S*≠{}
  shows *IArray.length* (*sum f S*) = *Max* {*IArray.length* (*f x*)| *x*. *x* ∈ *S*}
  ⟨*proof*⟩

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

lemma *length-zero-iarray*: *IArray.length 0* = *0*
  ⟨*proof*⟩

lemma *minus-zero-iarray*:
  fixes *A*::'*a*::{*group-add*} *iarray*
  shows *A* − *0* = *A*
⟨*proof*⟩

10.2 Inner mult over real iarrays

definition *inner-iarray* :: *real iarray* => *real iarray* => *real* (**infixl** ‹·i› *70*)
  where *inner-iarray A B* = *sum* (λ*n*. *A* !! *n* ∗ *B* !! *n*) {*0*..<*IArray.length A*}

lemma *vec-to-iarray-inner*:
  a · *b* = *vec-to-iarray a* ·*i vec-to-iarray b*
⟨*proof*⟩

lemma *vec-to-iarray-scaleR*:
  vec-to-iarray (*a* $*_R$ *x*) = *a* $*_R$ (*vec-to-iarray x*)
  ⟨*proof*⟩

10.3 Gram Schmidt over IArrays

definition *Gram-Schmidt-column-k-iarrays A k*
  = *tabulate2* (*nrows-iarray A*) (*ncols-iarray A*) (λ*a b*. (*if b* = *k*
  then (*column-iarray b A* − *sum* (λ*x*. (((*column-iarray b A*) ·*i x*) / (*x* ·*i x*)) $*_R$
x)
  (*set* (*List.map* (λ*n*. *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*
⟨*proof*⟩

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*
  ⟨*proof*⟩

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*)
  ⟨*proof*⟩

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_)
  (λ_a 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* = (*matrix-to-iarray* (*fst X*), *matrix-to-iarray* (*snd X*))

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

12 Examples of execution using floats and IArrays

theory *Examples-QR-IArrays-Float*
imports
QR-Decomposition-IArrays
HOL−Library.Code-Real-Approx-By-Float
begin

12.1 Examples

definition *example1* = (*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* (*divide-by-norm A*)))

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

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

definition *example4* = (*let A* = *list-of-list-to-matrix* [[*1*,*2*,*4*],[*9*,*4*,*5*],[*0*,*0*,*4*]]::*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* = *list-of-list-to-matrix* [[*1*,*sqrt 2*,*4*],[*sqrt 5*,*4*,*5*],[*0*,*sqrt 7*,*4*]]::*real^3^3 in*
iarray-of-iarray-to-list-of-list (*matrix-to-iarray* (*fst* (*QR-decomposition A*))))

definition *example6* = (*let A* = *list-of-list-to-matrix* [[*1*,*sqrt 2*,*4*],[*sqrt 5*,*4*,*5*],[*0*,*sqrt 7*,*4*]]::*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 (λi. show-real (v !! i)) (IArray.length v)*

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

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

lemma *matrix-to-iarray-show-matrix-real[code-unfold]: matrix-to-iarray (show-matrix-real v)*
= show-matrix-real-iarrays (matrix-to-iarray v)
⟨*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 ∘ show-vec-real-iarrays*

definition *print-mat-aux A = IArray.of-fun (λi. print-vec-mat (A !! i)) (IArray.length A)*

definition *print-mat = IArray.list-of ∘ print-mat-aux ∘ 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^3^4 in rank A = ncols A*

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

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

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

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^3^4*
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^3^4 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]]::real^3^4 in rank A = ncols A*

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

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

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

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

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

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

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

value *let A = list-of-list-to-matrix*
[[1,−0.6691],[1,−0.3907],[1,−0.1219],[1,0.3090],[1,0.5878]]::real^2^5;
b = list-to-vec [0.3704,0.5,0.6211,0.8333,0.9804]::real^5;

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

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

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

end

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

theory *Generalizations2*
imports
Rank-Nullity-Theorem.Fundamental-Subspaces
begin

14.1 Conjugate class

class *cnj* = *field* +
fixes *cnj* :: *'a⇒'a*
assumes *cnj-idem*[*simp*]: *cnj* (*cnj a*) = *a*
and *cnj-add*: *cnj* (*a*+*b*) = *cnj a* + *cnj b*
and *cnj-mult*: *cnj* (*a* ∗ *b*) = *cnj a* ∗ *cnj b*
begin

lemma *two-not-one*: *2* ≠ (*1*::*'a*)
⟨*proof*⟩

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*⇒*real*)= *id*

instance
⟨*proof*⟩
end

instantiation *complex* :: *cnj*
begin

definition (*cnj-complex* :: *complex*⇒*complex*)= *Complex.cnj*

instance
⟨*proof*⟩
end

14.2 Real_of_extended class

class *real-of-extended* = *real-vector* + *cnj* +
 fixes *real-of* :: *'a* ⇒ *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*)≥*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* (λ*i*. *f i*) *A*) = *sum* (λ*i*. *real-of* (*f i*)) *A*
⟨*proof*⟩

end

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

definition *real-of-real* :: *real* ⇒ *real* **where** *real-of-real* = *id*

instance
⟨*proof*⟩
end

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

definition *real-of-complex* :: *complex* ⇒ *real* **where** *real-of-complex* = *Re*

instance
⟨*proof*⟩
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* ⇒ *'b* ⇒ *'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* ≤ *real-of* (*inner x x*)
and *inner-eq-zero-iff* [*simp*]: *inner x x* = *0* ⟷ *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*) ⟷ *x* ≠ *0*

interpretation *RV-inner*: *inner-product-space scaleR inner*
⟨*proof*⟩

interpretation *RR-inner*: *inner-product-space scaleR* (∗)
⟨*proof*⟩

interpretation *CC-inner*: *inner-product-space* ((∗)::*complex*⇒*complex*⇒*complex*)
λ*x y*. *x*∗*cnj y*
⟨*proof*⟩

context *inner-product-space*
begin

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

lemma *inner-minus-left* [*simp*]: *inner* (− *x*) *y* = − *inner x y*
⟨*proof*⟩

lemma *inner-diff-left*: *inner* (*x* − *y*) *z* = *inner x z* − *inner y z*
⟨*proof*⟩

lemma *inner-sum-left*: *inner* ($\sum$ *x*∈*A*. *f x*) *y* = ($\sum$ *x*∈*A*. *inner* (*f x*) *y*)
⟨*proof*⟩

Transfer distributivity rules to right argument.

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

lemma *inner-scaleR-right* [*simp*]: *inner x* (*scale r y*) = (*cnj r*) ∗ (*inner x y*)
⟨*proof*⟩

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

lemma *inner-minus-right* [*simp*]: *inner x* (− *y*) = − *inner x y*
⟨*proof*⟩

lemma *inner-diff-right*: *inner x* (*y* − *z*) = *inner x y* − *inner x z*
⟨*proof*⟩

lemma *inner-sum-right*: *inner x* ($\sum$ *y*∈*A*. *f y*) = ($\sum$ *y*∈*A*. *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* ≤ *real-of* (*inner x x* + (*cnj a*) ∗ (*inner x y*) + *a* ∗ ((*cnj* (*inner x y*)) + (*cnj a*) ∗ (*inner y y*)))
⟨*proof*⟩

lemma *real-inner-inner*: *real-of* (*inner x x* ∗ *inner y y*) = *real-of* (*inner x x*) ∗

real-of (*inner y y*)
⟨*proof*⟩

lemma *Cauchy-Schwarz-ineq*:
real-of (*cnj* (*inner x y*) $*$ *inner x y*) $\leq$ *real-of* (*inner x x*) $*$ *real-of* (*inner y y*)
⟨*proof*⟩
end

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

context *inner-product-space*
begin

definition *norm x* = (*sqrt* (*real-of* (*inner x x*)))

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

lemma *inner-cnj-ge-zero*[*simp*]: *real-of* ((*inner x y*) $*$ *cnj* (*inner x y*)) $\geq$ *0*
⟨*proof*⟩

lemma *power2-norm-eq-inner*: $(norm\ x)^2$ = *real-of* (*inner x x*)
⟨*proof*⟩

lemma *Cauchy-Schwarz-ineq2*:
sqrt (*real-of* (*cnj* (*inner x y*) $*$ *inner x y*)) $\leq$ *norm x* $*$ *norm y*
⟨*proof*⟩

end

14.3.2 Orthogonality

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

context *inner-product-space*
begin

definition *orthogonal x y* $\longleftrightarrow$ *inner x y* = *0*

lemma *orthogonal-clauses*:
orthogonal a 0
orthogonal a x $\Longrightarrow$ *orthogonal a* (*scale c x*)
orthogonal a x $\Longrightarrow$ *orthogonal a* ($-$ *x*)
orthogonal a x $\Longrightarrow$ *orthogonal a y* $\Longrightarrow$ *orthogonal a* (*x* + *y*)
orthogonal a x $\Longrightarrow$ *orthogonal a y* $\Longrightarrow$ *orthogonal a* (*x* $-$ *y*)
orthogonal 0 a
orthogonal x a $\Longrightarrow$ *orthogonal* (*scale c x*) *a*
orthogonal x a $\Longrightarrow$ *orthogonal* ($-$ *x*) *a*
orthogonal x a $\Longrightarrow$ *orthogonal y a* $\Longrightarrow$ *orthogonal* (*x* + *y*) *a*
orthogonal x a $\Longrightarrow$ *orthogonal y a* $\Longrightarrow$ *orthogonal* (*x* $-$ *y*) *a*

⟨*proof*⟩

lemma *inner-commute-zero*: $(inner\ xa\ x = 0) = (inner\ x\ xa = 0)$
⟨*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 $\bigwedge y.\ y \in S \Longrightarrow orthogonal\ x\ y$
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* :: $'a\widehat{\ }'n \Rightarrow\ 'a\widehat{\ }'n \Rightarrow 'a$
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*: $real\text{-}of\ (inner\ x\ x) *_R A = inner\ x\ x * A$
⟨*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* λ*x y*. *x* ∗ *cnj y inner-cols*
\+ *ROWS?*: *vec-real-inner* λ*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*. ∀ *y* ∈ *W*. *orthogonal y x*}

lemma *subspace-orthogonal-complement*: *subspace* (*orthogonal-complement W*)
⟨*proof*⟩

lemma *orthogonal-complement-mono*:
assumes *A-in-B*: *A* ⊆ *B*
shows *orthogonal-complement B* ⊆ *orthogonal-complement A*
⟨*proof*⟩

lemma *B-in-orthogonal-complement-of-orthogonal-complement*:
shows *B* ⊆ *orthogonal-complement* (*orthogonal-complement B*)
⟨*proof*⟩

end

14.7 Generalizing projections

context *inner-product-space*
begin

Projection of two vectors: v onto u

definition *proj v u* = *scale* (*inner v u* / *inner u u*) *u*

Projection of a onto S

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

lemma *vector-sub-project-orthogonal-proj*:
  shows *inner b* (*x* − *proj x b*) = *0*
⟨*proof*⟩

lemma *orthogonal-proj-set*:
  assumes *yC*: *y*∈*C* **and** *C*: *finite C* **and** *p*: *pairwise orthogonal C*
  shows *orthogonal* (*a* − *proj-onto a C*) *y*
⟨*proof*⟩

lemma *pairwise-orthogonal-proj-set*:
  assumes *C*: *finite C* **and** *p*: *pairwise orthogonal C*
  shows *pairwise orthogonal* (*insert* (*a* − *proj-onto a C*) *C*)
  ⟨*proof*⟩
end

lemma *orthogonal-real-eq*: *RV-inner.orthogonal* = *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*]: (χ *i j*. *cnj* ((χ *i j*. *cnj* (*A* \$ *i* \$ *j*)) \$ *i* \$ *j*)) = *A*
  ⟨*proof*⟩

lemma *cnj-transpose*[*simp*]: (χ *i j*. *cnj* (*transpose A* \$ *i* \$ *j*)) = *transpose* (χ *i j*. *cnj* (*A* \$ *i* \$ *j*))
  ⟨*proof*⟩

lemma *null-space-orthogonal-complement-row-space*:
  fixes *A*::′*a*^′*cols*::{*finite*, *wellorder*}^′*rows*::{*finite*, *wellorder*}
  shows *null-space A* = *COLS.v.orthogonal-complement* (*row-space* (χ *i j*. *cnj* (*A* \$ *i* \$ *j*)))
⟨*proof*⟩

lemma *left-null-space-orthogonal-complement-col-space*:
fixes $A::'a^{\frown\prime}cols::\{finite,\ wellorder\}^{\frown\prime}rows::\{finite,\ wellorder\}$
shows $left\text{-}null\text{-}space\ A = ROWS.v.orthogonal\text{-}complement\ (col\text{-}space\ (\chi\ i\ j.\ cnj\ (A\ \$\ i\ \$\ j)))$
$\langle proof \rangle$

end

We can get the explicit results for complex and real matrices

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

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

lemma *null-space-orthogonal-complement-row-space-complex*:
fixes $A::complex^{\frown\prime}cols::\{finite,wellorder\}^{\frown\prime}rows::\{finite,wellorder\}$
shows $null\text{-}space\ A = complex\text{-}matrix.orthogonal\text{-}complement\ (row\text{-}space\ (\chi\ i\ j.\ cnj\ (A\ \$\ i\ \$\ j)))$
$\langle proof \rangle$

lemma *left-null-space-orthogonal-complement-col-space-complex*:
fixes $A::complex^{\frown\prime}cols::\{finite,\ wellorder\}^{\frown\prime}rows::\{finite,\ wellorder\}$
shows $left\text{-}null\text{-}space\ A = complex\text{-}matrix.orthogonal\text{-}complement\ (col\text{-}space\ (\chi\ i\ j.\ cnj\ (A\ \$\ i\ \$\ j)))$
$\langle proof \rangle$

lemma *null-space-orthogonal-complement-row-space-reals*:
fixes $A::real^{\frown\prime}cols::\{finite,wellorder\}^{\frown\prime}rows::\{finite,wellorder\}$
shows $null\text{-}space\ A = real\text{-}matrix.orthogonal\text{-}complement\ (row\text{-}space\ A)$
$\langle proof \rangle$

lemma *left-null-space-orthogonal-complement-col-space-real*:
fixes $A::real^{\frown\prime}cols::\{finite,\ wellorder\}^{\frown\prime}rows::\{finite,\ wellorder\}$
shows $left\text{-}null\text{-}space\ A = real\text{-}matrix.orthogonal\text{-}complement\ (col\text{-}space\ A)$
$\langle proof \rangle$

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

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

15.1.2 General properties about *sum-list*

lemma *sum-list-remdups*:
assumes $!!i\ j.\ i<\mathit{length}\ xs \wedge j<\mathit{length}\ xs \wedge i \neq j$
$\wedge\ xs\ !\ i = xs\ !\ j \longrightarrow xs\ !\ i = 0 \wedge xs\ !\ j = 0$
shows $\mathit{sum\text{-}list}\ (\mathit{remdups}\ xs) = \mathit{sum\text{-}list}\ xs$
⟨*proof*⟩

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

15.1.3 Proving a code equation to improve the performance

lemma *set-map-column*:
$\mathit{set}\ (\mathit{map}\ (\lambda n.\ \mathit{column}\ (\mathit{from\text{-}nat}\ n)\ G)\ [0..<\mathit{to\text{-}nat}\ b]) = \{\mathit{column}\ i\ G|i.\ i<b\}$
⟨*proof*⟩

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

lemma *column-Gram-Schmidt-upt-k-repeated-0'*:
  fixes *A*::*real*^′*n*::{*mod-type*}^′*m*::{*mod-type*}
  assumes *i-not-k*: *i*≠*j* **and** *ij*: *i*<*j* **and** *j*: *j*≤*from-nat k*
  and *c-eq*: *column j* (*Gram-Schmidt-upt-k A k*)
  = *column i* (*Gram-Schmidt-upt-k A k*)
  and *k*: *k*<*ncols A*
  shows *column j* (*Gram-Schmidt-upt-k A k*) = *0*
  ⟨*proof*⟩

lemma *column-Gram-Schmidt-upt-k-repeated-0*:
  fixes *A*::*real*^′*n*::{*mod-type*}^′*m*::{*mod-type*}
  assumes *i-not-k*: *i*≠*j* **and** *ij*: *i*<*j* **and** *j*: *j*≤*k*
  and *c-eq*: *column j* (*Gram-Schmidt-upt-k A* (*to-nat k*))
  = *column i* (*Gram-Schmidt-upt-k A* (*to-nat k*))
  shows *column j* (*Gram-Schmidt-upt-k A* (*to-nat k*)) = *0*
  ⟨*proof*⟩

corollary *column-Gram-Schmidt-upt-k-repeated*:
  fixes *A*::*real*^′*n*::{*mod-type*}^′*m*::{*mod-type*}
  assumes *i-not-k*: *i*≠*j* **and** *ij*: *i*≤*k* **and** *j*≤*k*
  and *c-eq*: *column j* (*Gram-Schmidt-upt-k A* (*to-nat k*))
    = *column i* (*Gram-Schmidt-upt-k A* (*to-nat k*))
  shows *column j* (*Gram-Schmidt-upt-k A* (*to-nat k*)) = *0*
  and *column i* (*Gram-Schmidt-upt-k A* (*to-nat k*)) = *0*
⟨*proof*⟩

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

lemma *Gram-Schmidt-upt-k-efficient-induction*:
  fixes *A*::*real*^′*n*::{*mod-type*}^′*m*::{*mod-type*}
  assumes *Gram-Schmidt-upt-k A k* = *foldl Gram-Schmidt-column-k-efficient A* [*0*..<*Suc k*]
  and *suc-k*: *Suc k* < *ncols A*
  shows *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 $A::real^{\frown}{'}n::\{mod\text{-}type\}^{\frown}{'}m::\{mod\text{-}type\}$
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 algoritm, 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 $A::real^{\frown}{'}n::\{mod\text{-}type\}^{\frown}{'}m::\{mod\text{-}type\}$
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*) (λ*a b. let column-b-A = column-iarray b A in*
(*if b = k then* (*column-b-A* − *sum-list* (*map* (λ*x.* ((*column-b-A* $\cdot$*i x*) / (*x* $\cdot$*i x*)) $*_R$ *x*)
((*List.map* (λ*n. 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 (*tabulate2 nr nc f*) !! *i* !! *j* = *f i j*
⟨*proof*⟩

lemma *vec-to-iarray-minus*[*code-unfold*]:
vec-to-iarray (*a* − *b*) = (*vec-to-iarray a*) − (*vec-to-iarray b*)
⟨*proof*⟩

lemma *vec-to-iarray-minus-nth*:
assumes *A*: *i*<*IArray.length* (*vec-to-iarray A*)
and *B*: *i*<*IArray.length* (*vec-to-iarray B*)
shows (*vec-to-iarray A* − *vec-to-iarray B*) !! *i*
= *vec-to-iarray A* !! *i* − *vec-to-iarray B* !! *i*
⟨*proof*⟩

lemma *sum-list-map-vec-to-iarray*:
assumes *xs* ≠ []
shows *sum-list* (*map* (*vec-to-iarray* ∘ *f*) *xs*) = *vec-to-iarray* (*sum-list* (*map f xs*))
⟨*proof*⟩

15.2.3 Proving the equivalence

lemma *matrix-to-iarray-Gram-Schmidt-column-k-efficient*:
fixes *A*::*real*^'*n*::{*mod-type*}^'*m*::{*mod-type*}
assumes *k*: *k*<*ncols A*
shows *matrix-to-iarray* (*Gram-Schmidt-column-k-efficient A k*)
= *Gram-Schmidt-column-k-iarrays-efficient* (*matrix-to-iarray A*) *k*
⟨*proof*⟩

lemma *matrix-to-iarray-Gram-Schmidt-upt-k-efficient*:
fixes *A*::*real*^'*n*::{*mod-type*}^'*m*::{*mod-type*}
assumes *k*: *k*<*ncols A*
shows *matrix-to-iarray* (*Gram-Schmidt-upt-k A k*)
= *foldl Gram-Schmidt-column-k-iarrays-efficient* (*matrix-to-iarray A*) [*0* ..<*Suc k*]
⟨*proof*⟩

lemma *matrix-to-iarray-Gram-Schmidt-matrix-efficient*[*code-unfold*]:
fixes *A*::*real*^'*n*::{*mod-type*}^'*m*::{*mod-type*}
shows *matrix-to-iarray* (*Gram-Schmidt-matrix A*)
= *Gram-Schmidt-matrix-iarrays-efficient* (*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