Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

A formalisation in HOL of the Fundamental Theorem of Linear Algebra and its application to the solution of the least squares problem

Jesús Aransay · Jose Divasón

the date of receipt and acceptance should be inserted later

Abstract In this paper we show how a thoughtful reusing of libraries can provide concise proofs of non-trivial mathematical results. Concretely, we formalise in Isabelle/HOL a proof of the *Fundamental Theorem of Linear Algebra* for vector spaces over *inner product spaces*, the *Gram-Schmidt* process of orthogonalising vectors over $\mathbb{R}$, its application to get the QR decomposition of a matrix, and the *least squares approximation* of systems of linear equations without solution, in a modest number of lines (*ca.* 2,700). This work intensively reuses previous results, such as the *Rank-Nullity Theorem* and various applications of the *Gauss-Jordan algorithm*. The formalisation is also accompanied by code generation and refinements that enable the execution of the presented algorithms in Isabelle and SML.

Keywords Least squares problem · QR decomposition · Interactive Theorem Proving · Linear Algebra · Code Generation · Symbolic Computation

1 Introduction

Interactive theorem proving is a field in which impressive problems are being challenged and overcome successfully (recently, the Odd Order Theorem [27] and the Flyspeck project, that reached the formalisation of the proof of the

J. Aransay
Departamento de Matemáticas y Computación
Universidad de La Rioja
Edif. Luis Vives, c/ Luis de Ulloa n. 2
E-mail: jesus-maria.aransay@unirioja.es

J. Divasón
Departamento de Matemáticas y Computación
Universidad de La Rioja
Edif. Luis Vives, c/ Luis de Ulloa n. 2
E-mail: jose.divasonm@unirioja.es

Kepler Conjecture [32], and, few years ago, the seL4 operating system kernel [39]). Still, new challenges usually require an accordingly impressive previous infrastructure to succeed (for instance, the SSReflect extension created for the Four-Colour Theorem and the Odd Order Theorem, and the Simpl imperative language and the AutoCorres translator which are keystones in the seL4 verification). This infrastructure, once developed, shall be reusable and applicable enough to overcome new challenges. Even if some developments or libraries reach a status of "keystones", and new projects are regularly built on top of them, this design principle does not always hold. For instance, the Isabelle theorem prover offers a repository of developments, whose inputs are refereed and later on maintained, called Archive of Formal Proofs (or AFP). Blanchette *et al.* [10] presented a survey about the reutilisation of such entries. Directly quoting the conclusions of this survey, "There is too little reuse to our taste; the top 3 articles are reused 9, 6 and 4 times."; we assume that these conclusions can be spread to theorem provers as a whole. In this paper we present various pieces of work which take great advantage of previously developed tools (either well-established parts of the Isabelle Library or 10 existing developments of the AFP) to fulfil in an affordable number of lines a complete work in Linear Algebra.

This work can be divided into four different parts. Firstly, we introduce the formalisation of the result called by Strang the *Fundamental Theorem of Linear Algebra* (see [49,50]), which establishes the relationships between the dimensions and bases of the four fundamental subspaces (the row space, column space, null space, and left null space) associated to a given linear map between two finite-dimensional vector spaces. This theorem is also closely tied to the notion of *orthogonality* of subspaces, whose implementation we explore in the interactive theorem prover Isabelle [44], and more concretely starting from its HOL Multivariate Analysis Library [35] (*HMA* in the sequel). The notion is already present in the library, but related concepts such as the projection of a vector onto a subspace are not. Secondly, we formalise the Gram-Schmidt process, which permits to obtain an *orthogonal set of vectors* from a given set of vectors. Gram-Schmidt possesses remarkable features, such as preserving the spanning set of collections of vectors and providing *linearly independent* vectors, whose formalisation we present. Thirdly, as a natural application of the Gram-Schmidt process we implement the QR decomposition of a matrix into the product of two different matrices (the first one containing an *orthonormal* collection of vectors, the second one being upper triangular). We formalise the relevant properties of this decomposition reusing some of the previous work that we created in a formalisation of the Gauss-Jordan algorithm [5, 21]. Fourthly, we formalise the application of the QR decomposition of a matrix to compute the *least squares approximation* to an unsolvable system of linear equations, exploiting some of the infrastructure for *norms* and distances in the HMA Library, and reusing the computation of solutions to systems of linear equations provided by *Gauss-Jordan*.

Finally, and based on our previous infrastructure for the Gauss-Jordan algorithm, we present examples of execution of the least squares problem (which

internally uses Gram-Schmidt and QR decomposition) inside Isabelle and also from the code generated to SML. More particularly, taking advantage of an existing development in Isabelle for *symbolically* computing with $\mathbb{Q}$ extensions of the form $\mathbb{Q}[\sqrt{b}]$ [51], exact symbolic computations of the QR decomposition, and thus of the least squares approximation of systems of linear equations are performed. We also present some optimisations performed over the original algorithm to improve its performance. Consequently, this work completes our previous developments [4, 5, 21] where the computation of the solution to systems of linear equations was formalised (for systems with unique solution, multiple solutions and without exact solution), computing also the *least squares approximation* to systems without solution.

In Linear Algebra, the QR decomposition is of interest by itself because of the insight that it offers about the vector subspaces of a linear map (it provides orthonormal bases of the four fundamental subspaces). We point the interested reader to the textbook by Strang [50, Chap. 4] for further information and applications. The method of *least squares* is usually attributed to Gauss, who already had a solution to it in 1795 that successfully applied to predict the orbit of the asteroid Ceres in 1801 (see [9]).

The paper will be organised as follows. In Section 2 we introduce the mathematical notions involved in our development by means of their representation in the HMA Library. In Section 3 we introduce the Fundamental Theorem of Linear Algebra. In Section 4 we present our formalisation of the Gram-Schmidt process. Section 5 presents the formalisation of the QR decomposition. Section 6 shows the application of the QR decomposition to the *least squares approximation*. In Section 7 we describe code generation of the aforementioned algorithms, introduce some code optimisations to improve performance, and present some examples of execution. Section 8 presents a brief survey of related formalisations. Finally, in Section 9 we present some conclusions of the completed work. The development is available from the Isabelle Archive of Formal Proofs [23]. Since the development relies upon 10 AFP articles, we recommend the interested reader to download the complete version of the AFP.

2 Preliminaries

2.1 Mathematical context

Matrices (over fields) are the representation in Linear Algebra of *linear maps* between *finite-dimensional* vector spaces. Therefore, let f be $f\colon F^m \to F^n$ and $A \in M_{(n,m)}(F)$ the matrix representing f with respect to suitable bases of F^m and F^n. The properties of A provide relevant information about f. For instance, computing the dimension of the *range* of f (or the *rank* or dimension of the *column space* of A, $\{A\,y \mid y \in F^m\}$ in set notation), or the dimension of its *kernel* (or the *null space* of A, $\{x \mid A\,x = 0\}$) we can detect if f is either *injective* or *surjective*. The *Rank-Nullity Theorem*, that we formalised for vector spaces over a generic field in a previous work [20], states that, for a linear

map f, the dimension of its range plus the one of its kernel amounts to the dimension of the source vector space (m in our case). The *reduced row echelon form* (or rref) of a matrix, that can be computed by means of the *Gauss-Jordan algorithm*, gives the dimension of both subspaces and their corresponding bases (we presented a formalisation of this algorithm in a previous work [5, 21]).

Associated to every matrix (and linear map), another two different subspaces exist. They are named *row space* ($\{A^T y \mid y \in F^n\}$ in set notation, where A^T denotes the transpose matrix of A) and *left null space* ($\{x \mid A^T x = 0\}$ in set notation). The four subspaces (usually named *four fundamental subspaces*) together share interesting properties about their dimensions and bases, that tightly connect them. These connections provide valuable insight to study systems of linear equations $A\,x = b$.

The aforementioned subspaces and results also have a *geometrical* interpretation that requires introducing a new operation, the *inner product* of vectors ($\langle\cdot,\cdot\rangle : V \times V \to F$, for a vector space V over a field F, being F either $\mathbb{R}$ or $\mathbb{C}$), which satisfies the following properties:

- $\langle x, y\rangle = \overline{\langle y, x\rangle}$, where $\overline{\langle\cdot,\cdot\rangle}$ denotes the conjugate;
- $\langle ax, y\rangle = a\langle x, y\rangle$, $\langle x + y, z\rangle = \langle x, z\rangle + \langle y, z\rangle$;
- $\langle x, x\rangle \geq 0$, $\langle x, x\rangle = 0 \Rightarrow x = 0$.

Note that in the particular case of the finite-dimensional vector space $\mathbb{R}^n$ over $\mathbb{R}$, the inner or *dot product* of two vectors $u, v \in \mathbb{R}^n$ is defined as $u \cdot v = \sum_{i=1}^{n} u_i v_i$. When $F = \mathbb{R}$, the *conjugate* is simply the identity.

Then, two vectors are said to be *orthogonal* when their inner product is 0 (which geometrically means that they are perpendicular). The *row space* and the *null space* of a given matrix are *orthogonal complements*, and so are the *column space* and the *left null space*. These results are brought together in the *Fundamental Theorem of Linear Algebra*, whose statement we include in Section 3. Thanks to the *orthogonality* of the row and null spaces, every vector $x \in \mathbb{R}^m$ can be decomposed as $x = x_r + x_n$, where x_r belongs to the row space of A and x_n belongs to the null space of A, and therefore $x_r \cdot x_n = 0$. Now, $A\,x = A\,(x_r + x_n)$ and this is equal to $A\,x_r$ (this hint will be crucial for the *least squares approximation* of systems of linear equations that we describe in Section 6).

2.2 Isabelle - HOL Multivariate Analysis Library

In this work we use various tools and developments carried out on top of Isabelle [44]. The keystone of our work is the HMA Library [35], that we briefly introduce in this section. Other tools which are also crucial for this work, such as code generation, code refinements, and an Isabelle development that permits to implement and compute with field extensions of the form $\mathbb{Q}[\sqrt{b}]$ will be succinctly presented when they show up in our work.

The HMA Library is a set of Isabelle theories introducing mathematical definitions and results including disciplines such as Topology, Algebra, and

Analysis. They are vastly inspired by the work of Harrison [33, 34] in HOL Light, which he used as a basis to complete proofs of the Fundamental Theorem of Algebra and of the Stone-Weierstrass Theorem (and later on as a basis of the Flyspeck project [32]). The translation of this Library from HOL Light to Isabelle/HOL is far from complete. It is mainly being done *by hand* and, apparently, translating HOL Light tactics and proofs to Isabelle is quite intricate. Paulson, Hölzl, Eberl, Himmelmann, Heller, Immler, and ourselves have or are actively contributing to this translation, and also to extend the HMA Library in other directions (such as the ones presented in this paper and in our previous works [5, 6]).

In its Isabelle version, it intensively uses the implementation of *type classes* in the system by Haftmann and Wenzel [31] to represent mathematical structures (such as semigroups, rings, fields and so on), a representation with various relevant features, such as enabling the possibility of operator overloading, the definition of concrete instances of type classes, and the code generation of instances of these structures. A type class definition can be seen as a logical specification of the set of types that contains the operations and satisfies the stated properties. We recommend the work by Hölzl, Immler, and Huffman [38] for a thorough description of the type classes appearing in the HMA Library. As an example of a type class (part of the HMA Library) that is widely used in our work we present the definition of *vector spaces* over $\mathbb{R}$.

```
class real_vector = scaleR + ab_group_add +
  assumes "scaleR a (x + y) = scaleR a x + scaleR a y"
  and "scaleR (a + b) x = scaleR a x + scaleR b x"
  and "scaleR a (scaleR b x) = scaleR (a * b) x"
  and "scaleR 1 x = x"
```

One particular limitation of the library, in both its HOL Light and Isabelle versions, is that it is mainly designed to deal with structures over $\mathbb{R}$, even if most of its notions admit further generalisation. For instance, our formalisation of the *Gauss-Jordan* algorithm over generic fields, based on the library representation of matrices, led us to generalise some of these concepts. We presented this generalisation to fields, sometimes direct, but some other times intricate in a previous work [6].

As an example, we present the *generalised* definition of *vector space*, which requires a different Isabelle mechanism than type classes (*locales*), since its definition involves two different type variables (the field, that in `real_vector` was the fixed type `real`, and the Abelian group).

```
locale vector_space =
 fixes scale::"'a::field ⇒ 'b::ab_group_add ⇒ 'b"
 assumes "scale a (x + y) = scale a x + scale a y"
 and "scale (a + b) x = scale a x + scale b x"
 and "scale a (scale b x) = scale (a * b) x"
 and "scale 1 x = x"
```

This definition of *vector space* is already part of the Isabelle Library. Unfortunately, some other derived mathematical structures and definitions (con-

jugate, inner product spaces, orthogonality) have not been generalised yet in the HMA Library. Despite that being a sensible work, it would involve the reformulation of several results in those new introduced structures, depriving us from one of the features of our work (the reutilisation of previous developments). In Section 3 we will compare the complexity (in number of code lines) of building our work on top of the HMA Library (using real inner product spaces) and that of building from scratch a new theory on inner product spaces; we have chosen as a touchstone the proof of the Fundamental Theorem of Linear Algebra.

Both the *Gram-Schmidt* algorithm and the QR decomposition are built upon the notion of *orthogonality*. This notion requires a new type class based upon vector spaces, named in the HMA Library `real_inner` (which describes an inner product space over the real numbers). It introduces an *inner* or dot product, which is then used to define the *orthogonality* of vectors.

```
context real_inner
begin
 definition "orthogonal x y ⟷ x · y = 0"
end
```

Some results on this work require *finite-dimensional* real vector spaces. Therefore, we make use of the *Euclidean space* type class (also part of the HMA Library), which is derived from `real_inner` by fixing a finite orthonormal basis. As a particularly interesting instance of the *Euclidean space* type class, $\mathbb{R}^n$ (for every n finite) is provided in the HMA Library. The representation of n-dimensional vectors, with n a finite type, due to Harrison, is a crucial aspect of the HMA Library (note that the underlying type system of the HOL family of provers, such as HOL Light and Isabelle/HOL, excludes dependent types, and consequently the possibility of defining n-dimensional vectors depending directly on a natural number, n). Its type definition and particular notation follows.

```
typedef ('a, 'b) vec = "UNIV :: ('b::finite) ⇒ 'a) set"
  morphisms vec_nth vec_lambda ..
notation vec_nth (infixl "$" 90) and vec_lambda (binder "χ" 10)
```

Vectors in finite dimensions are represented by means of *functions* from an underlying finite type to the type of the vector elements. The Isabelle constant `UNIV` denotes the set of every such function. Indeed, **typedef** builds a new type as a subset of an already existing type (in this particular case, the set includes every function whose source type is finite). Elements of the newly introduced type and the original one can be *converted* by means of the introduced **morphisms**. The **notation** clause introduces an infix notation (`$`) for converting elements of type `vec` to functions and a binder χ that converts functions to elements of type `vec`.

The definition becomes specially relevant since matrices will be also represented as an iterated construction over vectors (and thus matrices are represented as functions from a finite type to a type of vectors). In the sequel,

we replace `('a, 'b) vec` by the notation `'a^'b`, which is reminiscent of LaTeX notation.

The `real` data type in Isabelle is internally represented by means of equivalence classes of Cauchy sequences over the rational numbers. In our formalisation we abstract over these implementation details, and work exclusively with the data type properties (mainly, that the type is a *field*). By sticking to the real numbers, we also avoid some intricacies (for instance, the *conjugate* operation for real numbers happens to be the identity).

The aforementioned notions are the ones used in the implementation and formalisation of our objects of study (the Fundamental Theorem of Linear Algebra, Gram-Schmidt orthogonalisation, QR decomposition, and the least squares approximation). The `finite` type class over which the type definition `vec` ranges represents types with a *finite* universe; these *finite* sets have an inductive definition and also an induction rule which is used to perform inductive proofs over vectors indexes. When required, we impose an additional restriction over types used to represent vector indexes, which is that of belonging to the type class `wellorder`;[1] our particularly defined type class `mod-type` allows us to prove properties of matrices by induction on their rows or columns, and can be replaced by the types $\mathbb{Z}_n$ where $n \in \mathbb{N}$ (which are proven instances of this type class) in the code execution process.

3 The Fundamental Theorem of Linear Algebra

Theorem 1 *Fundamental Theorem of Linear Algebra. Let $A \in M_{(n,m)}(\mathbb{R})$ be a matrix and $r = \operatorname{rank} A$; then, the following equalities hold:*

1. *The dimensions of the column space and the null space of A are equal to r and $m - r$ respectively;*
2. *The dimensions of the row space and the left null space of A are equal to r and $n - r$ respectively;*
3. *The row space and the null space are orthogonal complements;*
4. *The column space and the left null space are orthogonal complements.*

Let us stress that items 1 and 2 also hold for $A \in M_{(n,m)}(F)$, with F a field, whereas items 3 and 4 hold for inner product spaces. We first introduce the notion of *span* (which is defined for generic *vector spaces*) for the sake of completeness. The `hull` binary (and infix) operator defines the resulting set of intersecting every `subspace` containing `S`:

```
context vector_space
begin
 definition "span (S::'b set) = subspace hull S"
end
```

[1] Every finite set can be equipped with a well-order, but they are represented by means of different type classes in the Isabelle library.

The definition of the fundamental subspaces in Isabelle reads as follows. Note the different kinds of product introduced in the definitions, including `op v*`, which stands for *vector times matrix*, `op *v`, for *matrix times vector*. Since we are out of the scope of `vector_space`, in the definitions of `row_space` and `col_space` we explicitly provide the operation `op *s` that fixes the underlying field and Abelian group (in other words, different scalar products could be used to deal simultaneously with different vector spaces):

```
definition left_null_space: "left_null_space A = {x. x v* A = 0}"
definition null_space: "null_space A = {x. A *v x = 0}"
definition row_space: "row_space A = vector_space.span op *s (rows A)"
definition col_space: "col_space A = vector_space.span op *s (columns A)"
```

The definitions of the *row* and *column* spaces are proven equivalent to the ones given in Section 2.1:

```
lemma fixes A :: "'a::field^'b::{finite,wellorder}^'c::{finite,wellorder}"
  shows "row_space A = {w. ∃y. transpose A *v y = w}"
  and "col_space A = {y. ∃x. A *v x = y}"
```

Item 1 in Theorem 1 is usually labelled as the *Rank-Nullity Theorem*, and we completed its formalisation generalised to matrices over fields (see [20]). From an existing basis of the null space and its completion up to a basis of F^m, it is proven that the dimension of the column space is equal to r. Then, the column space is proved equal to the range by means of algebraic rewriting.

In order to prove item 2 in Theorem 1, we apply again the *Rank-Nullity Theorem* to A^T. Additionally, it must be proven that the dimension of the row space is equal to the rank of A. This particular proof, for matrices over generic fields, involves the computation of the reduced row echelon form (or *rref*) of A, and it requires reusing our formalisation of the *Gauss-Jordan algorithm* [5,21]. The key idea is to prove that elementary row operations preserve both the row rank and the column rank of A, and then to compare the row and column ranks of the rref of A, concluding that they are equal. We describe this proof in [6]. Up to now, proofs have been carried out in full generality (for matrices over a generic field F).

Items 3 and 4 in Theorem 1 claim that the *row space* and the *null space* of a given linear map are orthogonal complements, and so are the *column space* and the *left null space*. The *orthogonal complement* of a subspace W is the set of vectors of V orthogonal to every vector in W (note that, following the HMA Library, the notion of orthogonality already places us in *inner product spaces over* $\mathbb{R}$):

```
definition "orthogonal_complement W = {x. ∀y∈W. orthogonal x y}"
```

Since the definition of the null space claims that this space is equal to the x such that $A\,x = 0$ and the row space of A is the one generated by the rows of A, both spaces are proven to be *orthogonal*; a similar reasoning over A^T proves that the left null space and the column space are also *orthogonal*.

```
lemma fixes A::"real^'cols::{finite,wellorder}^'rows::{finite,wellorder}"
  shows "left_null_space A = orthogonal_complement (col_space A)"
    and "null_space A = orthogonal_complement (row_space A)"
```

Note that the *Rank-Nullity Theorem* is the key result to prove that the fundamental subspaces are *complementary*. The definitions and proofs introduced in this section can be found in the files *Fundamental_ Subspaces* and *Least_ Squares_ Approximation* (where the *Rank-Nullity Theorem* is already incorporated to the development) of our development [23]. Thanks to the intensive reuse of these definitions and results, the complete proof of Theorem 1 took us 80 lines of Isabelle code.

As a matter of experiment, we tried to generalise the notion of *inner product space over* $\mathbb{R}$ to that of *inner product space over a field* F, and then replay the proof of Theorem 1. This generalisation can be found in file *Generalizations2* of our development [23]. The number of lines devoted to define the required notions, state Theorem 1 and prove it in full generality was *ca.* 650. Being the generalisation of the results presented in this work to inner product spaces over a field a sensible and interesting work, we stick in this work to *inner product spaces over* $\mathbb{R}$ since this decision gives us the chance to reuse the libraries in the HMA Library, instead of starting from scratch and reproducing them in full generality.

4 A formalisation of the Gram-Schmidt algorithm

In this section we introduce the Gram-Schmidt process that leads to the computation of an orthogonal basis of a vector space and its formalisation. Let us note that *orthonormal* vectors are *orthogonal* vectors whose norm is equal to 1. Another relevant concept is the *projection* of a vector v onto a vector u, and that of the projection onto a set. The Isabelle definitions follow; `setsum` denotes the result of applying the operation `(λx. proj a x)` to every element of `S` and computing their sum (these definitions can be found in file *Projections* of our development [23] together with some of their relevant properties):

```
definition "proj v u = (v · u / (u · u)) *R u"
definition "proj_onto a S = setsum (λx. proj a x) S"
```

The Gram-Schmidt process takes as input a (finite) set of vectors $\{v_1 \dots v_k\}$ (which need not be linearly independent, neither be a set with size smaller than or equal to the dimension of the underlying vector space) and iteratively subtracts from each of them their *projection* onto the previous ones. The process can be implemented in Isabelle as follows. The definition `Gram_Schmidt_step` takes a vector a and a list of vectors ys and subtracts from a its projections onto the vectors of ys. The obtained vector will be *orthogonal* to every vector in the input list ys. Note that we have replaced *sets of vectors* by *lists of vectors* for simplicity (`op @` denotes the appending operation on lists); a similar process could be applied to finite indexed sets:

```
definition "Gram_Schmidt_step a ys = ys @ [a - proj_onto a (set ys)]"
```

This *step* is *folded* over a list of vectors xs and the empty list, obtaining thus a list of vectors whose projections onto each other are 0 (*i.e.*, are orthogonal).

```
definition "Gram_Schmidt xs = foldr Gram_Schmidt_step xs []"
```

The defined function `Gram_Schmidt` satisfies two properties. First, the vectors in its output list must be *pairwise orthogonal*:

```
lemma Gram_Schmidt_pairwise_orthogonal:
  fixes xs::"('a::{real_inner}^'b) list"
  shows "pairwise orthogonal (set (Gram_Schmidt xs))"
```

Second, the *span* of the sets associated to both the output and input lists must be equal (note that here the definition `real_vector.span` does not require the underlying scalar product, as it was the case with `vector_space.span`):

```
lemma Gram_Schmidt_span:
  fixes xs::"('a::{real_inner}^'b) list"
  shows "real_vector.span (set (Gram_Schmidt xs)) = real_vector.span (set xs)"
```

The proofs of the properties `Gram_Schmidt_pairwise_orthogonal` and `Gram_Schmidt_span` are carried out by induction over the input list. Under these two conditions, whenever the input list is a basis of the vector space, the output list will also be a basis (the predicate `distinct` is used to assert that there are not repeated vectors in the input and output lists).

```
corollary orthogonal_basis_exists':
  fixes V :: "(real^'b) list"
  assumes B: "is_basis (set V)" and d: "distinct V"
  shows "is_basis (set (Gram_Schmidt V))
  ∧ distinct (Gram_Schmidt V) ∧ pairwise orthogonal (set (Gram_Schmidt V))"
```

A well-known variant of the Gram-Schmidt process is the *modified Gram-Schmidt* process; given a set of vectors $\{v_1 \dots v_k\}$, instead of subtracting from a vector v_i (where $1 \leq i \leq k$) its projections onto all its predecessors $(v_1 \dots v_{i-1})$, as Gram-Schmidt does, it subtracts from a vector v_i its projection onto u_{i-1}, where u_{i-1} is the result of orthogonalising v_{i-1}. In exact arithmetic, the modified Gram-Schmidt process produces the same result as the Gram-Schmidt process; from a numerical point of view, it reduces round-off errors, since each vector is orthogonalised with respect to another one. The formalisation of the modified process is similar to the one we have implemented. Since we do not aim at using numerical approximations, we have not implemented it.

As a previous step for the QR decomposition of matrices, we introduced a definition of the *Gram-Schmidt* process directly over the *columns of a matrix*. To get that, the above operation `Gram_Schmidt` could be applied to the list of columns of the matrix (indeed, that was our first version), but that would require two conversions between matrices and lists. Instead, in order to improve

efficiency, we have preferred to build a new matrix from a function, using the χ binder (the morphism defining a `vec` from a function).

The operation `Gram_Schmidt_column_k` returns a matrix where *Gram-Schmidt* is performed over column `k` and the remaining columns are not changed. This operation is then folded over the list of the input matrix columns. Note that `k` is a natural number, whereas rows and columns indexes are elements of the finite types introduced in Section 2.2, and thus the operation `from_nat` is applied to convert between them.

```
definition "Gram_Schmidt_column_k A k = (χ a 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..<k+1]"
definition "Gram_Schmidt_matrix A = Gram_Schmidt_upt_k A (ncols A - 1)"
```

The definition of `Gram_Schmidt_matrix` has been proven to satisfy similar properties to `Gram_Schmidt`. Additionally, both definitions have been set up to enable code generation and execution from Isabelle to both SML and Haskell.

The following expression can be now evaluated in Isabelle (note the use of intermediary functions for inputting a matrix as a list of lists and outputting the resulting matrix as a list of lists). In this setting, the function `Gram_Schmidt_matrix` is being evaluated. In Section 7 we will improve its performance using a refinement of these functions to *immutable arrays*:

```
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)"
```

The obtained result is:

```
"[[4,50/77,15/13,0], [-6,-75/77,10/13,0], [5,-130/77,0,0]]"
```

Note that the output vectors are *orthogonal*, but not *orthonormal*. We address this issue in the next section, when formalising the QR decomposition. The formalisations presented in this section are available in the file *Gram_ Schmidt* from [23].

5 A formalisation of the QR decomposition algorithm

The QR decomposition of a matrix A is defined as a pair of matrices, $A = QR$, where Q is a matrix whose columns are *orthonormal* and R is an *upper triangular* matrix (which in fact contains the elementary column operations that have been performed over A to reach Q). The literature includes different variants of this decomposition (see for instance [9, Chapt. 3, 4]). More concretely, it is possible to distinguish two different decompositions of a given matrix $A \in M_{(m,n)}(\mathbb{R})$:

- If A is *full column rank*, A can be decomposed as QR, where $Q \in M_{(m,n)}(\mathbb{R})$ and its columns are orthonormal vectors, and $R \in M_{(n,n)}(\mathbb{R})$ is an upper triangular and invertible matrix.

– A can also be decomposed as QR, where $Q \in M_{(m,m)}(\mathbb{R})$ and is orthonormal, and $R \in M_{(m,n)}(\mathbb{R})$ is an upper triangular (but neither square, nor invertible) matrix. This case is called *full QR decomposition*.

In this work we formalise the first case, where the number of rows of A will be greater than or equal to the number of columns. Indeed, this is the version of the decomposition which is directly applicable to solve the *least squares problem*, as we explain in Section 6. In the particular case where A is not full column rank, we do not compute the QR decomposition, but, as we present in Section 6, we solve the problem by means of the *Gauss-Jordan* algorithm. Let us describe how the decomposition is performed.

Given a matrix $A = (a_1 \mid \ldots \mid a_n) \in M_{(m,n)}(\mathbb{R})$ (where $n \leq m$) whose columns a_i are linearly independent vectors, the matrix $Q \in M_{(m,n)}(\mathbb{R})$ is the matrix with columns $(q_1 \mid \ldots \mid q_n)$, where q_i is the normalised vector a_i minus its projections onto $q_1 \ldots q_{i-1}$ (and thus, *orthogonal* to both $a_1 \ldots a_{i-1}$ and $q_1 \ldots q_{i-1}$). The matrix $R \in M_{(n,n)}(\mathbb{R})$ can be expressed as $R = Q^T A$.

Once we have computed the *Gram-Schmidt* process over the vectors of a matrix in Section 4 (recall the Isabelle function `Gram_Schmidt_matrix`), and introducing an operation to *normalise* every column in a matrix, the computation of the QR decomposition is defined in Isabelle as follows:

```
definition "divide_by_norm A  = (χ a b. normalize (column b A) $ a)"
definition "QR_decomposition A =
 (let Q = divide_by_norm (Gram_Schmidt_matrix A) in (Q, (transpose Q) ** A))"
```

The literature suggests some other ways to compute the matrices Q and R, in such a way that the coefficients of matrix R are computed in advance, and then used in the computation of the columns of Q; see for instance the algorithms labelled as *Classical Gram-Schmidt* and *Modified Gram-Schmidt* by Björck [9, Chap. 2.4]. These algorithms avoid some unnecessary operations in our Isabelle formalisation (in particular, they avoid the computations of Q^T and the product $Q^T A$). Instead, our formalised version directly uses the output of the Gram-Schmidt orthogonalisation process presented in Section 4 and computes *a posteriori* the coefficients in R.

The properties of Q and R need to be proved. Once that in Section 4 we proved that the columns of the matrix computed with `Gram_Schmidt_matrix` are pairwise orthogonal and that they have a span equal to the one of the input matrix, these properties are straightforward to prove for Q. The property of the columns of Q having norm equal to 1 is proven also directly from the definition of Q. For its intrinsic interest we illustrate the property of Q and A having equal *column space*:

```
corollary col_space_QR_decomposition:
  fixes A::"real^'n::{mod_type}^'m::{mod_type}"
  defines "Q ≡ fst (QR_decomposition A)"
  shows "col_space A = col_space Q"
```

Another crucial property of Q (and Q^T) that is required later in the least squares problem is the following one (note that it is stated for possibly *non-square* matrices with more rows than columns):

```
lemma orthogonal_matrix_fst_QR_decomposition:
  fixes A::"real^'n::{mod_type}^'m::{mod_type}"
  defines "Q ≡ fst (QR_decomposition A)"
  assumes r: "rank A = ncols A"
  shows "transpose Q ** Q = mat 1"
```

This property is *commutative* for square matrices ($Q^TQ = QQ^T = I_n$ and thus $Q^{-1} = Q^T$) but it does not hold that $QQ^T = I_m$ for *non-square* ones. Its proof is completed by case distinction in the matrix indexes; being $Q = (q_1 \mid \cdots \mid q_n)$, and thus $Q^T = \begin{pmatrix} \frac{q_1}{\cdots} \\ q_n \end{pmatrix}$, when multiplying row i of Q^T (which is q_i) times column i of Q, the result is 1 since the vectors are *orthonormal*. On the contrary, when multiplying row i of Q^T (which is q_i) times column j of Q, the result is 0 because of *orthogonality*.

Then, the most relevant properties of R are being upper triangular and invertible. Indeed, being $A = (a_1 \mid \cdots \mid a_n)$, $R = Q^TA = \begin{pmatrix} q_1 \cdot a_1 & q_1 \cdot a_2 & \cdots \\ q_2 \cdot a_1 & q_2 \cdot a_2 & \cdots \\ \cdots & \cdots & \cdots \end{pmatrix}$. The following lemma proves the matrix R being upper triangular:

```
lemma upper_triangular_snd_QR_decomposition:
  fixes A::"real^'n::{mod_type}^'m::{mod_type}"
  defines "Q ≡ fst (QR_decomposition A)" and "R ≡ snd (QR_decomposition A)"
  assumes r: "rank A = ncols A"
  shows "upper_triangular R"
```

The matrix R is also *invertible*:

```
lemma invertible_snd_QR_decomposition:
  fixes A::"real^'n::{mod_type}^'m::{mod_type}"
  defines "Q ≡ fst (QR_decomposition A)" and "R ≡ snd (QR_decomposition A)"
  assumes r: "rank A = ncols A"
  shows "invertible R"
```

The properties satisfied by the QR decomposition (in this statement, of non-square matrices) can be finally stated in a single result (the result for square matrices of size n also proves the columns of Q being a *basis* of $\mathbb{R}^n$). The result sums up the properties of Q and R that have been formalised along Sections 4 and 5:

```
lemma QR_decomposition:
  fixes A::"real^'n::{mod_type}^'m::{mod_type}"
  defines "Q ≡ fst (QR_decomposition A)" and "R ≡ snd (QR_decomposition A)"
  assumes r: "rank A = ncols A"
  shows "A = Q ** R ∧
         pairwise orthogonal (columns Q) ∧ (∀i. norm (column i Q) = 1) ∧
         (transpose Q) ** Q = mat 1 ∧ vec.independent (columns Q) ∧
         col_space A = col_space Q ∧ card (columns A) = card (columns Q) ∧
         invertible R ∧ upper_triangular R"
```

The formalisations carried out in this section are available in the file *QR_ Decomposition* from our development [23].

6 Solution of the least squares problem

The previous decomposition can be used for different applications. In this work we focus on finding the best approximation of a system of linear equations *without solution*. In this way, we complete our previous work [5], in which the computation of the solution of systems of linear equations was formalised thanks to the *Gauss-Jordan elimination*.

The *best approximation* of a system $Ax = b$, in this setting, means to find the elements $\hat{x}$ such that minimise $\|e\|$, where $e = A\hat{x} - b$. Our aim is to prove that $\hat{x}$ is the solution to $A\hat{x} = \hat{b}$, where $\hat{b}$ denotes the projection of b onto the column space of A. The solution for the general case (also known as the *rank deficient case*) is usually performed by means of the *Singular Value Decomposition* (or SVD); this decomposition provides, for any real or complex matrix A, three matrices U, Σ, V such that $A = U\Sigma V^H$ (where V^H is the result of conjugating each element of V and then transposing the matrix, and $\Sigma = \left(\begin{smallmatrix} \Sigma_1 & 0 \\ 0 & 0 \end{smallmatrix}\right)$, where $\Sigma_1 = \text{diag}\,(\sigma_1 \dots \sigma_r)$ and σ_i denote the singular values of A, in such a way that $A = \sum_{i=1}^{n} \sigma_i u_i v_i^H$).

The existence of the SVD decomposition of a matrix can be proven by induction without particular difficulties (see [9, Th. 1.2.1]). On the contrary, the computation of the SVD decomposition (see, for instance, [9, Sect. 2.6]) requires the computation of eigenvalues and eigenvectors of matrices, whose computation requires numerical methods. In this work we solve the case where the input matrix A of the system $A\,x = b$ is *full column rank* by means of the QR decomposition, and the general case will be solved applying the Gauss-Jordan algorithm.

We define the characterisation of the least squares approximation of a system as follows (following [9, Th. 1.1.2]):

```
definition "set_least_squares_approximation A b =
   {x. ∀y. norm (b - A *v x) ≤ norm (b - A *v y)}"
```

Prior to showing the utility of the QR decomposition to solve the previous problem, we prove that the closest point to a point $v \notin S$ in a subspace S (being X an orthogonal basis of S) is its projection onto that subspace:

```
lemma least_squares_approximation:
  fixes X::"'a::{euclidean_space} set"
  assumes "real_vector.subspace S" and "real_vector.independent X"
  and "X ⊆ S" and "S ⊆ real_vector.span X"
  and "pairwise orthogonal X"
  and "proj_onto v X ≠ y"
  and "y ∈ S"
  shows "norm (v - proj_onto v X) < norm (v - y)"
```

The lemma `least_squares_approximation`, states that the projection of b onto the range of A (that we denote by $\hat{b}$) is the closest point to b in this subspace. Let $\hat{x}$ be such that $A\,\hat{x} = \hat{b}$. Thanks to Theorem 1, $b - A\,\hat{x}$ belongs to the orthogonal complement of range A, which happens to be the *left null space*. Consequently, we know that the solutions to the least squares problem must satisfy the equation $A^T(b - A\,x) = 0$ (the converse also holds). From this property, the standard characterisation of the set of least squares approximations is obtained [9, Th. 1.1.2]:

```
lemma in_set_least_squares_approximation_eq:
  fixes A::"real^'cols::{finite,wellorder}^'rows"
  defines "A_T == transpose A"
  shows "(x ∈ set_least_squares_approximation A b) = (A_T ** A *v x = A_T *v
b)"
```

The proof of lemma `least_squares_approximation` makes use of the Pythagorean Theorem of real inner product spaces, whose proof we include because of its intrinsic interest[2] (it reproduces the conventional argument $||x + y||^2 = x \cdot x + x \cdot y + y \cdot x + y \cdot y$, which because of the orthogonality of x and y is equal to $x \cdot x + y \cdot y = ||x||^2 + ||y||^2$):

```
lemma Pythagorean_theorem_norm:
  assumes o: "orthogonal x y" shows "norm (x+y)^2=norm x^2 + norm y^2"
proof -
  have "norm (x+y)^2 = (x+y) · (x+y)" unfolding power2_norm_eq_inner ..
  also have "... = ((x+y) · x) + ((x+y) · y)" unfolding inner_right_distrib ..
  also have "... = (x · x) + (x · y) + (y · x) + (y · y) "
    unfolding real_inner_class.inner_add_left by simp
  also have "... = (x · x) + (y · y)" using o unfolding orthogonal_def
    by (metis comm_monoid_add_class.add.right_neutral inner_commute)
  also have "... = norm x^2 + norm y^2" unfolding power2_norm_eq_inner ..
  finally show ?thesis .
qed
```

Once we have characterised the set of least squares approximations, we distinguish whether A is full column rank or not:

– If A is not full column rank, A^TA does not have inverse, and the solution to the least squares problem can be obtained by applying the Gauss-Jordan algorithm (that we formalised in a previous work [5]) to the system $A^TA\hat{x} = A^Tb$ [9, Eq. 1.1.15]. Our Gauss-Jordan implementation would compute a single solution of the system plus a basis of the null space of A^TA:

```
lemma in_set_least_squares_approximation_eq:
  fixes A::"real^'cols::{finite,wellorder}^'rows"
  defines "A_T ≡ transpose A"
  shows "(x ∈ set_least_squares_approximation A b) = (A_T ** A *v x = A_T
*v b)"
```

[2] The proof can be completed in one single line of Isabelle code, but we usually favour Isar human-readable proofs [54].

– Otherwise, A^TA is an invertible matrix, and $\hat{x} = (A^TA)^{-1}A^Tb$. In this case, the solution is *unique*, and the set in the right hand side is a singleton. The following result proves the uniqueness and the explicit expression of the solution [9, Eq. 1.1.16]:

```
lemma in_set_least_squares_approximation_eq_full_rank:
  fixes A::"real^'cols::mod_type^'rows::mod_type"
  defines "A_T ≡ transpose A"
  assumes r: "rank A = ncols A"
  shows "(x ∈ set_least_squares_approximation A b) =
           (x = matrix_inv (A_T ** A) ** A_T *v b)"
```

As it may be noticed, the solution to the *least squares problem* does not demand the QR decomposition of A. The decomposition is used when A is an (full column rank) almost singular matrix (*i.e.*, its condition number, σ_1/σ_r, where σ_1 and σ_r are the greatest and smallest singular values of A, is "big", and the computation of $(A^TA)^{-1}$ seriously compromises floating-point precision). Even if numerical methods are not central to our aim, we point the interested reader to the works by Björck [9, Sect. 1.4] or [17, Sect. 2.4].

Since $\hat{x} = (A^TA)^{-1}A^Tb$, and using that A can be decomposed as QR, with Q a matrix of orthonormal vectors, and R upper triangular, one also has that $A^T = R^TQ^T$ (note that $Q^TQ = I$). Then, $\hat{x} = (R^TQ^TQR)^{-1}R^TQ^Tb$, and this equation can be reduced to $\hat{x} = R^{-1}Q^Tb$. Now, the matrices Q and R are obtained through the Gram-Schmidt process, and the inverse of R, which is upper triangular, can be performed by backward substitution. The Isabelle statement of this new equality follows [9, Th. 1.3.3]:

```
corollary in_set_least_squares_approximation_eq_full_rank_QR2:
  fixes A::"real^'cols::{mod_type}^'rows::{mod_type}"
  defines "Q ≡ fst (QR_decomposition A)" and "R ≡ snd (QR_decomposition A)"
  assumes r: "rank A = ncols A"
  shows "(x ∈ set_least_squares_approximation A b) =
           (x = matrix_inv R ** transpose Q *v b)"
```

The formalisations of the results in this section are available in the file *Least_Squares_Approximation* from our development [23]. In the next section we show how the previous results can be used to *compute* the least squares approximation of a linear system.

7 Code generation from the development

Up to now we have proved that given a matrix $A \in M_{(m,n)}(\mathbb{R})$ and a system of linear equations $Ax = b$ without solution there exists one or multiple least squares approximations to that system, and we have also provided and proved explicit expressions to identify them. In the case where A is not full column rank, computing the approximations requires solving the system $A^TA\hat{x} = A^Tb$; when A is full column rank, the approximation can be directly computed by means of the expression $\hat{x} = R^{-1}Q^Tb$.

The computation of the approximations, based on the previous expressions, requires various features.

- First, the underlying *real* type (and the required operations) needs an *executable* representation.
- Then, the representation (and the operations) chosen for matrices needs an *executable* version.
- For the case where A is not full column rank, an executable version of the Gauss-Jordan algorithm applied to compute the solution of systems of linear equations needs to be provided.

The first point admits various solutions. The default Isabelle type for `real` numbers is implemented by means of Cauchy sequences of rational numbers [12], but then code generation is contemplated for the subset of the real numbers formed by rational numbers. With this particular subset, there exists the possibility of executing inside of Isabelle arithmetic operations (implemented by means of rewrite rules) over elements of type `real`, as long as the results can be represented as quotients of rational numbers. This approach is valid, for instance, for executing Gauss-Jordan (even in this particular case, computing the Gauss-Jordan form of a random 15×15 matrix takes 3 minutes, even if this time heavily depends on the size of the coefficients), as long as matrices inputs are rational numbers, but it is not valid for computing QR decompositions.

Therefore, our alternative solution consists in making use of the aforementioned Isabelle code generation facility [28, 30] that translates Isabelle specifications to specifications in various functional languages (in this development, we make use exclusively of SML). The type *real* is generated by default in SML to quotients of integer numbers. Unfortunately, square roots computations are not possible in this setting (only *Gram-Schmidt* could be executed).

The code generator can alternatively be set up to *identify* (or serialise) an Isabelle type to SML native types; following this methodology, we used an already existing serialisation in the Isabelle library that maps the type *real* and its operations to the SML structure *Real* (and its underlying type *real*). This serialisation allows us to *compute* in SML the formalised algorithm of the least squares problem, but computations fall in the conventional rounding errors of double-precision floating-point numbers (despite the original algorithm being formalised, the computations cannot be trusted). File *Examples_QR_IArrays_Float* in our development [23] contains some examples of computations in SML following this methodology.

Fortunately, as we were completing this work, an Isabelle development named "Implementing field extensions of the form $\mathbb{Q}[\sqrt{b}]$" by Thiemann was published in the Isabelle Archive of Formal Proofs [51]. This development provides, among many other interesting features, a data type refinement for real numbers of the form $p + q\sqrt{b}$ (with $p, q \in \mathbb{Q}$, $b \in \mathbb{N}$, with $\sqrt{b}$ irrational, and b a prime product). The refinement can represent any real number in a field extension $\mathbb{Q}[\sqrt{b}]$; binary operations are implemented as partial functions, so that operations over numbers belonging to different field extensions do raise

exceptions. This refinement is available for computations inside of Isabelle, and also for code generation to SML. We make use of this development, and obtain exact symbolic computations, as long as we restrict ourselves to matrices whose inputs are in $\mathbb{Q}$ (if we input matrices in $\mathbb{Q}[\sqrt{b}]$ their normalisation may belong to $\mathbb{Q}[\sqrt{b}][\sqrt{a}]$, that is out of the scope of the presented Isabelle development).

The second concern to obtain computations is the *representation of matrices*. The representation we have used along the formalisation relies on a type `vec` representing vectors (and then its iterated construction to represent matrices) which corresponds to functions over finite domains. Additionally, some Isabelle definitions and statements are restricted to vectors indexed by the type class `mod_type`. These types and type classes are perfectly suited for code generation.

The following example shows the execution of the sum of a vector of rational numbers with itself; the `mod_type` class is replaced by an instance (the finite type with the three elements `0`, `1` and `2`, *i.e.* `3`). Operations to convert lists to vectors and *vice versa* are used for inputting and outputting the otherwise cumbersome elements of `vec` type:

```
value "(let
      A = list_of_list_to_matrix [[1/3,2,4/5],[9,4/7,5],[0,5/2,0]]::rat^3^3
      in matrix_to_list_of_list (A + A))"
```

The obtained output follows (it is obtained in Isabelle, and thus certified, based on a *rewriting* and *normalisation by evaluation* strategy [2, 30]):

```
"[[2 / 3, 4, 8 / 5], [18, 8 / 7, 10], [0, 5, 0]]"
```

The representation of vectors as functions (a given vector is internally represented as a lambda expression) can be improved by more convenient data types, such as *immutable arrays*. Accessing operations are then performed in constant time, whereas functions require being evaluated. Moreover, Isabelle offers an infrastructure for both data type and algorithmic refinement by Haftmann *et al.* [29]. This infrastructure permits to establish a *map* (or morphism) among data types (in our case, from vectors to immutable arrays), which, together with certain operations over immutable arrays, have to be proven equivalent to the ones that have been used in the formalisation. Isabelle offers some additional packages that further simplify the refinement of types and operations to *executable* counterparts (at least, *Lifting and Transfer* by Huffman and Kunčar [36], and the *Isabelle Refinement Framework* by Lammich [40]). None of them was applied in our case study. The first one, still ongoing work, could pay off in terms of code reusability among the different representations of vectors, but we are still exploring the possibilities of applying it to our setting. The latter is specially designed for dealing with *imperative programs* in which the underlying structures are maps or sets that are automatically converted to optimised representations. Despite its utility, since underlying types are vectors in our work, we could not directly apply the framework. In-

stead, we preferred to reuse the formalised link that we already developed and successfully tested in a previous work [4, 5].

In this work, we have reproduced in Isabelle the definitions of the QR decomposition for immutable arrays and proved their equivalence with respect to the vector versions. These lemmas are then used as rewrite rules (their left hand side is replaced by their right hand side) in the evaluation and code generation processes (and are named *code lemmas* in Isabelle jargon).

Nevertheless, some of the definitions over immutable arrays used in the Gauss-Jordan formalisation have been replaced in this development. For instance, *iarray addition*, which we defined in the Gauss-Jordan development as:[3]

```
plus_iarray A B =  IArray.of_fun (λn. A!!n + B!!n) (IArray.length A)
```

The infix operator `A!!n` denotes accessing to component `n` of `A`. The operation is not well-defined for values greater than or equal to `length A`. On the contrary, do note that the operation is not commutative (it is trivial to find counterexamples with vectors `B` whose length is greater than the one of `A`). In this development the operation `plus_iarray` is defined as:

```
plus_iarray A B = (let
        length_A = (IArray.length A); length_B = (IArray.length B);
        n = max length_A length_B ;
        A' = IArray.of_fun (λa. if a < length_A then A!!a else 0) n;
        B' = IArray.of_fun (λa. if a < length_B then B!!a else 0) n
    in IArray.of_fun (λa. A'!!a + B'!!a) n)
```

This new definition is *commutative*, and thus it permits to show that *iarrays* over a commutative monoid are an instance of the Isabelle type class `comm_monoid_add`. When proving commutativity, do note that it does not admit premises on the length of the vectors since it is the definition used to instantiate the type `IArray` with an operation `+` (or `plus`). Thanks to commutativity, several proofs involving *finite sums* (for instance, ranging over the columns of a matrix) of *iarrays* are simplified. On the other hand, this definition is more time consuming than the previous one, and it could have some impact on performance.

Some definitions along our development made use of operations that may be, *a priori*, non executable, such as `all` or `exists` applied to the elements of a variable of type `vec`. These definitions have to be restated over iarrays in a way that they are executable (in this particular case we take advantage of the `set` type constructor over lists giving place to sets that can be computed and traversed), and proved to be equivalent to the ones over `vec`.

```
fun all :: "('a ⇒ bool) ⇒ 'a iarray ⇒ bool"
  where "all p (IArray as) = (ALL a : set as. p a)"
fun exists :: "('a ⇒ bool) ⇒ 'a iarray ⇒ bool"
  where "exists p (IArray as) = (EX a : set as. p a)"
```

[3] Note that the type system takes care of the elements of type *vec* being of equal size and this assumption can then be avoided in the refinement to *immutable arrays*.

We then serialise these operations to corresponding ones on SML:

```
code_printing
  constant "IArray_Addenda_QR.exists" ⇀ (SML) "Vector.exists"
  | constant "IArray_Addenda_QR.all" ⇀ (SML) "Vector.all"
```

Then, definitions over `vec` type are proven *equivalent* (modulo type morphisms) to definitions over `iarray`:

```
definition "is_zero_iarray A =
    IArray_Addenda_QR.all (λi. A !! i = 0) (IArray[0..<IArray.length A])"

lemma is_zero_iarray_eq_iff:
  fixes A::"'a::{zero}^'n::{mod_type}"
  shows "(A = 0) = (is_zero_iarray (vec_to_iarray A))"
```

The previous set-up, together with the refinement of *real numbers* to *field extensions* $\mathbb{Q}[\sqrt{b}]$, gives place to the following symbolic computations of the matrices Q and R in Isabelle (computations are internally being performed in SML transparently to the user; they can also be internally performed in Isabelle). Do note that, once the refinement to *iarrays* has been performed, the operations internally being executed are the ones over *iarrays*:

```
definition "A ==
  list_of_list_to_matrix [[1,3/5,3],[9,4,5/3],[0,0,4],[1,2,3]]::real^3^4"
value "print_mat (fst (QR_decomposition A))"
value "print_mat (snd (QR_decomposition A))"
```

The results obtained follow (their computation time in SML is 0.001 s.). We have reproduced the example in Mathematica® version 10.4 [43], obtaining a similar time.[4]

```
"[["1/83*sqrt(83)",    "4/4233*sqrt(8466)",  "95/65229*sqrt(130458)"],
  ["9/83*sqrt(83)", "-11/8466*sqrt(8466)", "-19/130458*sqrt(130458)"],
  [            "0",                   "0",     "3/1279*sqrt(130458)"],
  ["1/83*sqrt(83)", "91/8466*sqrt(8466)",  "-19/130458*sqrt(130458)"]]"

"[["sqrt(83)", "193/415*sqrt(83)",       "21/83*sqrt(83)"],
  [       "0", "7/415*sqrt(8466)", "418/12699*sqrt(8466)"],
  [       "0",                "0",   "2/153*sqrt(130458)"]]"
```

We can also compute the *least squares approximation* to systems of equations with no solution. As we mentioned in Section 6, when $A = QR$ is full column rank, solving this problem requires computing the *inverse* of the matrix R, and this is done thanks to the *Gauss-Jordan* algorithm that we already formalised [5]. An interesting situation shows up here, related to the use of $\mathbb{Q}[\sqrt{b}]$ extensions. The solution to the least squares problem $Ax = b$ can be computed as $\hat{x} = R^{-1}Q^Tb$.

[4] The benchmarks have been carried out in laptop with an Intel Core i5-3360M processor, 4 GB of RAM, PolyML 5.5.2-3 and Ubuntu 14.04.

Given a matrix A, the computation of the matrix Q may involve the use of field extensions $\mathbb{Q}[\sqrt{b}]$, where b could be different in each column. Then, the computation of $R = Q^T A$ gives place to an upper triangular matrix (with each row in a possibly different extension of $\mathbb{Q}[\sqrt{b}]$), whose inverse is computed by means of elementary row operations, based on our implementation of the Gauss-Jordan algorithm [5, 21].

The *least squares approximation* $\hat{x}$ of a system $A\,x = b$ is computed symbolically as shown in the following example (the operation `the` is the Isabelle/HOL implementation of Hilbert's ϵ definite operator, since we have used an *option type* to represent the partiality of the `inverse_matrix` operation; details are given in our Gauss-Jordan development [5]):

```
definition "b ≡ list_to_vec [1,2,3,sqrt(2)]::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))"
```

The computed solution is `"["12269/17906 - 10443/35812 * sqrt(2)", "-11840/8953 + 5900/8953 * sqrt(2)", "1605/2558 - 57/5116 * sqrt(2)"]"`.

As we illustrate with the following computation, being $\hat{b} = A\,\hat{x}$, the difference $b - \hat{b}$ lies on the *left null space* of A, and therefore $A^T(b - \hat{b}) = 0$:

```
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 (transpose A *v (b - b2))"
```

Its output, as expected, is: `"["0", "0", "0"]"`.

We present the result of a problem related to the computation of the orbit of the comet Tentax [17, Ex. 1.3.4]. A brief statement of the problem follows. In a certain polar coordinate system, the following observations of the position of the comet were made:

r	2.70	2.00	1.61	1.20	1.02
θ	48°	67°	83°	108°	126°

Kepler's first law states that, neglecting the perturbations from planets, the comet should follow a plane hyperbolic or elliptic form. Therefore, the coordinates satisfy the equation:

$$r = \frac{p}{1 - e\cos\theta}$$

Where p denotes a parameter and e the eccentricity. If the relationship is rewritten as $1/p - (e/p)\cos\theta = 1/r$, it becomes linear in the parameters $x_1 = 1/p$ and $x_2 = e/p$. Then, the linear system $Ax = b$ is obtained, where:

$$A = \begin{pmatrix} 1 & -0.6691 \\ 1 & -0.3907 \\ 1 & -0.1219 \\ 1 & 0.3090 \\ 1 & 0.5878 \end{pmatrix}, \qquad b = \begin{pmatrix} 0.3704 \\ 0.5 \\ 0.6211 \\ 0.8333 \\ 0.9804 \end{pmatrix}.$$

The least squares problem is formulated in Isabelle as follows:

```
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)"
```

The obtained solution is `"["3580628725341/5199785740000", "251601193/519978574"]"` (corresponding to x_1 and x_2, from which the parameter p and the eccentricity e are computed). It is obtained in SML thanks to the refinement to iarrays. Computing time in SML is 0.0012 s., whereas in Mathematica® it is 0.0007 s.

As an additional example, we introduce *Hilbert matrices*, which are well-known for being ill-conditioned, and thus prone to round-off errors. Hilbert matrices are defined as:

$$H_{ij} = \frac{1}{i + j - 1}$$

For instance, the Hilbert matrix in dimension 6, H_6 has determinant equal to 1/186313420339200000 and the order of magnitude of its condition number is 10^7:

```
[ [ 1 ,1/2,1/3,1/4,1/5,1/6],
  [1/2,1/3,1/4,1/5,1/6,1/7],
  [1/3,1/4,1/5,1/6,1/7,1/8],
  [1/4,1/5,1/6,1/7,1/8,1/9],
  [1/5,1/6,1/7,1/8,1/9,1/10],
  [1/6,1/7,1/8,1/9,1/10,1/11] ]
```

We have computed the least squares approximation to the system $H_6\, x = (1\,0\,0\,0\,0\,0\,5)^T$ using the QR decomposition (this QR decomposition comprises coefficients of order 10^{19}). The least squares approximation of the system follows:

```
"["-13824","415170","-2907240","7754040","-8724240","3489948"]"
```

Its SML computation time is 0.013 s. or 0.022 s., depending on whether we use the optimisations presented in Section 7.1 or not. In Mathematica® the same computation takes 0.017 s.

7.1 Code Optimisations

The implementation of our first version of the QR algorithm admitted different types of performance optimisation that we also applied. Here we comment on three of them.

– First, there was an issue with the conversion from sets to lists in the code generation process. Let us recover the Isabelle definition introduced in Section 4, `proj_onto a S = setsum (λx. proj a x) S`. The definition applies an operation to the elements of a set `S` and then computes their sum. The Isabelle code generator is set up to refine sets to lists (whenever sets are finite), and thus the previous sum is turned into a list sum, by means of the following code equation (note that sums are abstracted to a generic "big operator" `F` defined for both sets or lists, and that we have omitted that the underlying structure is a commutative monoid):

```
lemma set_conv_list [code]:
 "set.F g (set xs) = list.F (map g (remdups xs))"
```

It is relevant to pay attention to the operation `remdups`; the input list `xs` that represents the set could contain duplicated elements, and therefore they have to be removed from that list for the equality to hold (for instance, the set $\{1, 2, 3\}$ can be originated by both $xs = [1, 2, 3, 3]$ and $xs = [1, 2, 3]$). When we applied code generation and by means of SML profiling techniques, we detected that `remdups` was one of the most time consuming operations in the QR executions. In our particular case, after applying `Gram_Schmidt_column_k` (this operation explicitly uses `proj_onto`, and hence `remdups`, see Section 4) to the first k columns of a set, the obtained columns are either 0 or orthogonal. In the second case, there are no duplicated columns. Interestingly, in the first case, there might be duplicated columns equal to 0; these columns, when used in later iterations of `Gram_Schmidt_column_k`, do not affect the final result. In any case (with the previous columns being 0 or orthonormal), the following operation, that avoids removing duplicates, is more efficient than `Gram_Schmidt_column_k`, and returns the same result when applied to the column $k+1$ of a matrix where `Gram_Schmidt_column_k` has been applied to the first k columns. The following definition (where `remdups` over the list of columns of the matrix has been avoided) is to be compared with the one of `Gram_Schmidt_column_k` presented in Section 4, that we repeat here to ease comparison:

```
definition "Gram_Schmidt_column_k A k =
   (χ a 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_column_k_efficient A k =
   (χ a b. (if b = from_nat k
   then (column b A - listsum (map (λx. ((column b A · x) / (x · x)) *R x)
    (map (λn. column (from_nat n) A) [0..<to_nat b]))
```

```
  else column b A) $ a)"
```

The proof of the equivalence between both definitions can be found in file *QR_Efficient* [23]; let us remark that the property only holds for a column $k+1$ when the first k columns have been already orthogonalised.
We have also used the standard strategy of providing code generation for sets as lists where duplicates are removed in the computation of inner products. By default, the inner product is computed over the *set* of indexes of the vectors (that are turned into lists to which `remdups` is applied, even when the set of indexes is known not to not contain repeated elements). The following code equation avoids this (in our case, unnecessary) check:

```
lemma [code]:
 "inner_iarray A B = listsum (map (λn. A!!n * B!!n) {0..<IArray.length A})"
```

– A second improvement on code performance was directly introduced by the Isabelle developers during the process of completing this work, and is related to the code generation set-up of the function `map_range`, that profiling showed as another bottleneck of our programs execution. The function `map_range` is internally used to apply a function to a range of numbers (and therefore it is being used, for instance, in our previous definitions `inner_iarray` or `Gram_Schmidt_column_k_efficient`):

```
definition map_range[code_abbrev]: "map_range f n = map f [0..<n]"
```

The original code equation for this definition follows:

```
lemma map_range_simps [simp, code]:
 "map_range f 0 = []"
 "map_range f (Suc n) = map_range f n @ [f n]"
```

This definition, in each iteration, builds two different lists and concatenates them. The operation can be completed over a single list, improving both memory usage and performance. The previous definition of `map_range` was removed from the Isabelle library on lists, and the conventional definition of map over lists used instead:

```
lemma [code]:
  "map f [] = []"
  "map f (x # xs) = f x # map f xs"
```

– Finally, we realised that the definition of `Gram_Schmidt` also had room for improvement, from a computational point of view. The definition of `Gram_Schmidt_column_k_iarrays_efficient` can be replaced by the following one (they are extensionally equal):

```
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 - listsum (map (λx. ((col_k ·i x) / (x ·i x)) *R x)
             ((map (λn. column_iarray n A) [0..<k]))))
  in (λa b. (if b = k then col else column_iarray b A) !! a))"
```

This definition makes use of `let` definitions to bind variables (such as `col_k`) that in `Gram_Schmidt_column_k_iarrays_efficient` where being performed (indeed, they were simply access operations) many times. Proving the equivalence between both definitions is straightforward (the proof is lemma `Gram_Schmidt_column_k_iarrays_efficient_eq` in file *QR_Efficient* of our development [23]).

The development also allows further computations, such as the projection of a vector onto a subspace, the *Gram-Schmidt* algorithm, *orthogonality* of vectors, solution of the least squares problem for matrices without full rank, and can be used to grasp the geometrical implications of the *Fundamental Theorem of Linear Algebra*. The previous computations and some other carried out with the refinement to floating-point numbers can be found in files *Examples_QR_Abstract_Symbolic*, *Examples_QR_IArrays_Symbolic*, *Examples_QR_Abstract_Float* and *Examples_QR_IArrays_Float* [23]. We formalised the computation of the bases of the four fundamental subspaces of a linear map in a previous work [5].

8 Related work

Abstract Algebra is a common topic for the theorem proving community. Some milestones in this field have already been pointed out in Section 1. Also the work by Harrison in HOL Light and its relation with the HMA Library has already been described, and used to our benefit. This seminal work in real vector spaces has been continued in some developments, such as the work by Afshar *et al.* [3], where complex vector spaces have also been formalised, following ideas and reusing definitions of real vector spaces, and applied to a case study of electromagnetic waves. However, we miss the definition of a "common place" or generic structure representing inner product spaces over real and complex numbers, such as the one introduced by us at the end of Section 3, that could permit a definition and formalisation of the Gram-Schmidt process for both structures simultaneously.

Some interesting works in Linear Algebra algorithms can also be found in the literature.

The Mizar Mathematical Library contains articles by various authors related to matrix computations. The most related one to the work we present here is a formalisation by Pąk of the Jordan Normal Form [46]; this work is not based on real numbers, but generically carried out for matrices over algebraically closed fields. Some articles for specifically complex matrices by Chang *et al.* [14,15] are also available. Basic definitions of operations (including inner product and conjugates) and their associated properties are introduced.

PVS also contains a remarkable library for real analysis (see for instance the work by Dutertre [24] and Butler [13]). Real numbers are axiomatised as a subtype of a generic *number* type. There is also a different representation of real numbers implemented by Lester [41], that uses Cauchy sequences, both of which are linked by means of intermediary results. Interval arithmetic has been then used to prove inequalities over the reals by Daumas, Lester, and Muñoz [18].

It is worth mentioning a recent development in Isabelle/HOL by Thiemann and Yamada [52], in which they have formalised the computation of the Jordan Normal Form. Their representation of matrices is slightly different from ours, since they use a generic type to represent *matrices* of any dimension, whereas ours has a hardwired representation of the size in the matrices types. Their choice enables them to apply the algorithm to input matrices whose dimension is unknown in advance, one of their prerequisites. On the contrary, they are forced to include premises in several lemmas fixing the dimensions for matrices before using them. Interestingly, they have been able to link their representation of matrices with ours (the one in the HMA Library) by means of the Lift and Transfer tools [36], and, for instance, reuse some of the results about determinants proven either in the HMA Library or in our work. They have also applied the refinement to *iarrays* that we proposed for the Gauss-Jordan algorithm [5]. Another interesting work in Isabelle based on the HMA Library representation of matrices is the one by Adelsberger *et al.* [1]. They prove the Cayley-Hamilton Theorem, but they do not pay attention to computability issues. In a later development of ours, where we formalise an algorithm computing the echelon form of a matrix over Euclidean domains [7, 22], we have refined their definitions of polynomial matrices, characteristic polynomial and so on, and used them to *evaluate* the Cayley-Hamilton Theorem statement in matrices over a generic field. The previous examples illustrate the applicability of our infrastructure and ideas to recent developments not completed by us.

Some works about inequalities with real numbers have also been carried out in Isabelle/HOL, such as the work by Hölzl [37]. In this work lower-bound and upper-bound inequalities of transcendental functions are formalised, and then used to compute intervals of real arithmetic expressions. The idea is to implement a process of *reification* of arithmetic expressions, and then a refinement of real numbers to floating-point numbers represented as a pair of integers for both the mantissa and exponent; the obtained floating-point numbers are then used to compute interval values for arithmetic expressions. These floating-point numbers are then generated to SML where inequalities are evaluated, and solved (when possible). The refinement has been successfully applied in Hölzl's case study, and also in the work of Obua and Nipkow [45] to formalise the basic linear programs emerging in the Flyspeck project. Even if both applications are devoted to the formalisation of inequalities (instead of solving equalities, as in our case study), the particular refinement of real numbers to floating-point numbers may be useful in our setting. Also as part of the Flyspeck project, Solovyev and Hales implemented several tools in the

HOL Light proof assistant in order to get formal verification of numerical computations [47, 48].

Some relevant works in the Coq proof assistant are worth mentioning, such as the CoqEAL (standing for Coq Effective Algebra Library) effort by Dénès *et al.* [19]. This work is devoted to develop a set of libraries and commodities over which algorithms over matrices can be implemented, proved correct, refined, and finally executed. In a previous work [5, Sect. 5.2] we presented a thorough comparison of our work and theirs, surveying also their most recent works [16]. We summarise the points made there. With respect to the *implementation and formalisation of algorithms*, the main difference relies on the possibility of using dependent types, and thus submatrices, in Coq. Incidentally, the QR decomposition is usually performed in a *per column* basis, so it is not clear in this particular case whether submatrices may simplify either the implementation or the proof of the algorithm. *Refinements* are also possible in CoqEAL, both at the level of data structures and algorithms; up to our knowledge, vectors and matrices are refined in CoqEAL, for execution purposes, to lists. Our Isabelle development uses immutable vectors, that theoretically seem a better performing option than lists. As we have illustrated in Section 7.1, we have also performed and formalised various algorithmic refinements inside of Isabelle. Finally, regarding *execution*, CoqEAL algorithms can be executed on top of the Coq virtual machine (and therefore *inside* of the Coq trusted kernel).

Also in Coq, Gonthier [26] implemented a version of Gaussian elimination (producing two different matrices, one describing the column space, the other one the row space, and a number, equal to the rank of the input matrix) that he later applied for the computation of several basic operations over linear maps; for instance, the computation of the four fundamental subspaces of a given matrix (we formalised similar computations [5, Sect. 5.2] by means of the Gauss-Jordan algorithm, and thus performing, a priori, a greater number of elementary operations than he does in Gaussian elimination), and also basic operations between subspaces (union, intersection, inclusion, addition). One of its most prominent features is the simplicity that he obtains in proofs of properties concerning the rank and the fundamental subspaces. Another one is the generality of the results presented there; since he aims at finally working with finite groups, most of the results are established in full generality (instead of restricted to reals, as we have assumed in several cases along this work). With respect to the work presented here, it seems that he has formalised neither inner product spaces nor orthogonality and the subsequent concepts (such as Gram-Schmidt, QR decomposition, and the least squares approximation). Since the focus of Gonthier's work seems to be in the formalisation of Linear Algebra, concerns about computability are neither tackled. Then, in an ongoing work based on SSReflect about the Discrete Fourier Transform, Gallego and Jouvelot [25] have proposed a representation of complex numbers over which they have implemented the inner product, norm, orthogonality and Hermitian transposes. This possibility illustrates that the generalisation of the HMA Library to more general structures (at least to the complex numbers)

is a feasible and sensible work (that also demands a careful design). The verification of floating-point mathematical computations and interval arithmetic have also been studied in Coq by means of the CoqInterval library, mainly developed by Melquiond [11, 42].

[11]

We are not aware of any previous formalisation of neither the QR decomposition nor the computation of the least squares approximation in any proof assistant.

9 Conclusions

This work can be considered as an advanced exercise in theorem proving, but at the same time it has required the use of well-established libraries (such as HMA) and the adaptation and set-up of some previous works. Some of these works had been completed by us (such as the Rank-Nullity theorem, the Gauss-Jordan development and the code refinements to *iarrays* and real numbers in Haskell) but some others tools were new to us (for instance, the code generator, *iarrays*, real numbers and the representation of field extensions). It is worth noting that our development relies on another 10 previous developments in the Isabelle AFP, two of them developed by the authors, and 8 of them from other authors. In that sense, with an affordable effort (the complete development sums up *ca.* 2,700 lines, plus 2,100 lines devoted to refinements, code generation, and examples), and also with a certain amount of previous knowledge of the underlying system, we have developed a set of results and formalised programs in Linear Algebra that corresponds to various lessons of an undergraduate text in mathematics (for instance, material which sums up 60 pages in the textbook by Strang [50, Sect. 3.6 and Chap. 4]). As a matter of comparison, the results presented in [50, Chap. 1 to Sect. 3.5], which include at least all the results that we already formalised in our previous work [5] and additional previous notions of the HMA Library, took us more than 15,000 lines of Isabelle code. From that point of view, we have to stress that this work would have been impossible without such previous developments; as it usually happens in mathematics, new results have to be built on top of established results. This principle is well-known in the formal methods community, but it is difficult to achieve; this work shows a successful case study where the principle has been materialised.

As further work, at least three possibilities show up. First, the results presented from Section 4 and on admit a generalisation from real inner product spaces to inner product spaces. The HMA Library would also benefit from such a generalisation. The possibility of generalising algebraic structures from the HMA Library has been already illustrated by our previous works with the Gauss-Jordan algorithm [5, 6] and also the echelon form [7, 22]. Second, the application of the QR decomposition to numerical problems such as the computation of the eigenvalues, and also the formalisation of the related Singular Value Decomposition would be of interest. Third, some experiments with code

generation to Haskell that we have performed recently [8] show that the refinement from the Isabelle *real* type to the Haskell type *Prelude.Rational* is slightly better performing than the one in SML to quotients of type *IntInt.int*; therefore, exploring the possibilities of code generation from the field extensions by Thiemann [51] to Haskell, instead of SML, using the type *Prelude.Rational* might be fruitful in improving the performance of the computations with the QR algorithm presented. Moreover, it would be desirable to study a recent work by Thiemann and Yamada [53] about a formalisation of algebraic numbers in Isabelle/HOL, which seems useful to get symbolic computations over more structures than field extensions of type $\mathbb{Q}[\sqrt{b}]$. Also in [8] we have explored the possibilities of refining the *real* Isabelle type to floating-point numbers in both SML and Haskell. Even if the obtained computations cannot be trusted, we show in that work that the precision obtained is orders of magnitude better than the one of the Gauss-Jordan algorithm with a similar refinement. The performance of the algorithm with floating-point numbers is also illustrated [8]. The possibility of formalising more stable versions of the Gram-Schmidt process (such as the modified Gram-Schmidt process) and studying their behaviour could be also interesting.

Acknowledgements The authors would like to thank the anonymous referees because of their valuable contributions along the review process. Particularly, their suggestions helped us to improve the overall clarity of the presentation and the related work section. This work has been partially supported by the research grant FPI-UR-12, from Universidad de La Rioja and by the project MTM2014-54151-P from Ministerio de Economía y Competitividad (Gobierno de España).

References

1. Adelsberger, S., Hetzl, S., Pollak, F.: The Cayley-Hamilton Theorem. Archive of Formal Proofs (2014). `http://afp.sf.net/entries/Cayley_Hamilton.shtml`, Formal proof development
2. Aehlig, K., Haftmann, F., Nipkow, T.: A Compiled Implementation of Normalization by Evaluation. Journal of Functional Programming **22**(1), 9–30 (2012)
3. Afshar, S.K., Aravantinos, V., Hasan, O., Tahar, S.: Formalization of Complex Vectors in Higher-Order Logic. In: S.M. Watt, J.H. Davenport, A.P. Sexton, P. Sojka, J. Urban (eds.) Intelligent Computer Mathematics: CICM 2014. Proceedings, *Lecture Notes in Artificial Intelligence*, vol. 8543, pp. 123–137. Springer (2014)
4. Aransay, J., Divasón, J.: Formalization and execution of Linear Algebra: from theorems to algorithms. In: G. Gupta, R. Peña (eds.) PostProceedings of the International Symposium on Logic-Based Program Synthesis and Transformation: LOPSTR 2013, *Lecture Notes in Computer Science*, vol. 8901, pp. 01 – 19. Springer (2014)
5. Aransay, J., Divasón, J.: Formalisation in higher-order logic and code generation to functional languages of the Gauss-Jordan algorithm. Journal of Functional Programming **25**, 1 – 21 (2015)
6. Aransay, J., Divasón, J.: Generalizing a Mathematical Analysis Library in Isabelle/HOL. In: K. Havelund, G. Holzmann, R. Joshi (eds.) NASA Formal Methods: NFM 2015, *Lecture Notes in Computer Science*, vol. 9508, pp. 415 – 421 (2015)
7. Aransay, J., Divasón, J.: Formalisation of the Computation of the Echelon Form of a Matrix in Isabelle/HOL. Formal Aspects of Computing (2016). Accepted for publication
8. Aransay, J., Divasón, J.: Verified Computer Linear Algebra. Accepted for publication in the Conference EACA 2016 (2016). `https://www.unirioja.es/cu/jearansa/archivos/vcla.pdf`

9. Björck, A.: Numerical methods for least squares problems. SIAM (1996)
10. Blanchette, J., Haslbeck, M., Matichuk, D., Nipkow, T.: Mining the Archive of Formal Proofs. In: M. Kerber (ed.) Conference on Intelligent Computer Mathematics: CICM 2015, *Lecture Notes in Computer Science*, vol. 9150, pp. 3–17. Springer (2015). Invited paper
11. Boldo, S., Jourdan, J., Leroy, X., Melquiond, G.: Verified Compilation of Floating-Point Computations. Journal of Automated Reasoning **54**(2), 135–163 (2015)
12. Boldo, S., Lelay, C., Melquiond, G.: Formalization of real analysis: a survey of proof assistants and libraries. Mathematical Structures in Computer Science **FirstView**, 1–38 (2016). DOI 10.1017/S0960129514000437. URL `http://journals.cambridge.org/article_S0960129514000437`
13. Butler, R.B.: Formalization of the Integral Calculus in the PVS Theorem Prover. Tech. Rep. NASA/TM-2004-213279, L-18391, NASA Langley Research Center (2004). `http://ntrs.nasa.gov/search.jsp?R=20040171869`
14. Chang, W., Yamazaki, H., Nakamura, Y.: A Theory of Matrices of Complex Elements. Formalized Mathematics **13**(1), 157–162 (2005). URL `http://fm.mizar.org/2005-13/pdf13-1/matrix_5.pdf`
15. Chang, W., Yamazaki, H., Nakamura, Y.: The Inner Product and Conjugate of Matrix of Complex Numbers. Formalized Mathematics **13**(4), 493–499 (2005). URL `http://fm.mizar.org/2005-13/pdf13-4/matrixc1.pdf`
16. Cohen, C., Dénès, M., Mörtberg, A.: Refinements for Free! In: G. Gonthier, M. Norrish (eds.) Certified Programs and Proofs: CPP 2013, *Lecture Notes in Computer Science*, vol. 8307, pp. 147–162. Springer (2013)
17. Dahlquist, G., Björck, A.: Numerical Methods in Scientific Computing. SIAM (2008)
18. Daumas, M., Lester, D., Muñoz, C.: Verified Real Number Calculations: A Library for Interval Arithmetic. IEEE Transactions On Computers **58**(2), 226 – 237 (2009)
19. Dénès, M., Mörtberg, A., Siles, V.: A refinement-based approach to Computational Algebra in COQ. In: L. Beringer, A. Felty (eds.) Interactive Theorem Proving: ITP 2012, *Lecture Notes in Computer Science*, vol. 7406, pp. 83–98. Springer (2012)
20. Divasón, J., Aransay, J.: Rank-Nullity Theorem in Linear Algebra. Archive of Formal Proofs (2013). `http://afp.sf.net/entries/Rank_Nullity_Theorem.shtml`
21. Divasón, J., Aransay, J.: Gauss-Jordan Algorithm and Its Applications. Archive of Formal Proofs (2014). `http://afp.sf.net/entries/Gauss_Jordan.shtml`, Formal proof development
22. Divasón, J., Aransay, J.: Echelon Form. Archive of Formal Proofs (2015). `http://afp.sf.net/entries/Echelon_Form.shtml`, Formal proof development
23. Divasón, J., Aransay, J.: *QR* Decomposition. Archive of Formal Proofs (2015). `http://afp.sf.net/entries/QR_Decomposition.shtml`, Formal proof development. Updated version available from `http://afp.sf.net/devel-entries/QR_Decomposition.shtml`
24. Dutertre, B.: Elements of Mathematical Analysis in PVS. In: J. von Wright, J. Grundy, J. Harrison (eds.) Theorem Proving in Higher Order Logics: TPHOLs 97, *Lecture Notes in Computer Science*, vol. 1125, pp. 141–156. Springer, Turku, Finland (1996)
25. Gallego-Arias, E.J., Jouvelot, P.: Adventures in the (not so) Complex Space. The Coq workshop 2015 (2015). `https://github.com/ejgallego/mini-dft-coq`
26. Gonthier, G.: Point-Free, Set-Free Concrete Linear Algebra. In: M. van Eekelen, H. Geuvers, J. Schmaltz, F. Wiedijk (eds.) Interactive Theorem Proving: ITP 2011, *Lecture Notes in Computer Science*, vol. 6898, pp. 103–118. Springer (2011)
27. Gonthier, G., Asperti, A., Avigad, J., Bertot, Y., Cohen, C., Garillot, F., Roux, S.L., Mahboubi, A., O'Connor, R., Biha, S.O., Pasca, I., Rideau, L., Solovyev, A., Tassi, E., Théry, L.: A Machine-Checked Proof of the Odd Order Theorem. In: S. Blanzy, C. Paulin-Mohring, D.Pichardie (eds.) Interactive Theorem Proving: ITP 2013, *Lecture Notes in Computer Science*, vol. 7998, pp. 163 – 179. Springer (2013)
28. Haftmann, F.: Code generation from Isabelle/HOL theories. `https://isabelle.in.tum.de/doc/codegen.pdf` (2016)
29. Haftmann, F., Krauss, A., Kuncar, O., Nipkow, T.: Data Refinement in Isabelle/HOL. In: S. Blazy, C. Paulin-Mohring, D. Pichardie (eds.) Interactive Theorem Proving: ITP 2013, *Lecture Notes in Computer Science*, vol. 7998, pp. 100 – 115. Springer (2013)

30. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: M. Blume and N. Kobayashi and G. Vidal (ed.) Functional and Logic Programming: FLOPS 2010, *Lecture Notes in Computer Science*, vol. 6009, pp. 103 – 117. Springer (2010)
31. Haftmann, F., Wenzel, M.: Constructive Type Classes in Isabelle. In: T. Altenkirch, C. McBride (eds.) Types for Proofs and Programs: TYPES 2006, *Revised Selected Papers*, *Lecture Notes in Computer Science*, vol. 4502, pp. 160–174. Springer (2007)
32. Hales, T., Adams, M., Bauer, G., Dang, D., Harrison, J., Hoang, T.L., Kaliszyk, C., Magron, V., McLaughlin, S., Nguyen, T.T., Nguyen, T.Q., Nipkow, T., Obua, S., Pleso, J., Rute, J., Solovyev, A., Ta, A.H.T., Tran, T.N., Trieu, D.T., Urban, J., Vu, K.K., Zumkeller, R.: A formal proof of the Kepler conjecture. `http://arxiv.org/abs/1501.02155` (2015)
33. Harrison, J.: A HOL Theory of Euclidean Space. In: J. Hurd, T. Melham (eds.) Theorem Proving in Higher Order Logics: TPHOLS 2005, *Lecture Notes in Computer Science*, vol. 3603, pp. 114 – 129. Springer (2005)
34. Harrison, J.: The HOL Light Theory of Euclidean Space. Journal of Automated Reasoning **50**(2), 173 – 190 (2013)
35. HOL Multivariate Analysis Library. `http://isabelle.in.tum.de/library/HOL/HOL-Multivariate_Analysis/index.html` (2016)
36. Huffman, B., Kunčar, O.: Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL. In: G. Gonthier, M. Norrish (eds.) Certified Programs and Proofs: CPP 2013, *Lecture Notes in Computer Science*, vol. 8307, pp. 131–146. Springer (2013)
37. Hölzl, J.: Proving Inequalities over Reals with Computation in Isabelle/HOL. In: G.D. Reis, L. Théry (eds.) International Workshop on Programming Languages for Mechanized Mathematics Systems: PLMMS'09, pp. 38–45. Munich (2009)
38. Hölzl, J., Immler, F., Huffman, B.: Type Classes and Filters for Mathematical Analysis in Isabelle/HOL. In: S. Blazy, C. Paulin-Mohring, D. Pichardie (eds.) Interactive Theorem Proving: ITP 2013, *Lecture Notes in Computer Science*, vol. 7998, pp. 279–294. Springer (2013)
39. Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: formal verification of an operating-system kernel. Commun. ACM **53**(6), 107–115 (2010)
40. Lammich, P.: Automatic Data Refinement. In: S. Blazy, C. Paulin-Mohring, D. Pichardie (eds.) Interactive Theorem Proving: ITP 2013, *Lecture Notes in Computer Science*, vol. 7998, pp. 84 – 99. Springer (2013)
41. Lester, D..R.: Real Number Calculations and Theorem Proving. In: O.A. Mohamed, C. Muñoz, S. Tahar (eds.) Theorem Proving in Higher Order Logics: TPHOLs 08, *Lecture Notes in Computer Science*, vol. 5170, pp. 215 – 229. Springer (2008)
42. Martin-Dorel, É., Melquiond, G.: Proving Tight Bounds on Univariate Expressions with Elementary Functions in Coq. Journal of Automated Reasoning pp. 1–31 (2015). DOI 10.1007/s10817-015-9350-4
43. Mathematica 10.4. Wolfram Research, Inc. Champaign, Illinois (2016)
44. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, *Lecture Notes in Computer Science*, vol. 2283. Springer (2002). Updated version available in `http://isabelle.in.tum.de/doc/tutorial.pdf`
45. Obua, S., Nipkow, T.: Flyspeck II: The Basic Linear Programs. Annals of Mathematics and Artificial Intelligence **56**, 245–272 (2009)
46. Pąk, K.: Jordan Matrix Decomposition. Formalized Mathematics **16**(4), 297–303 (2008). DOI 10.2478/v10037-008-0036-9
47. Solovyev, A., Hales, T.: Efficient formal verification of bounds of linear programs. In: Intelligent Computer Mathematics, *Lecture Notes in Computer Science*, vol. 6824, pp. 123–132. Springer (2011)
48. Solovyev, A., Hales, T.: Formal verification of nonlinear inequalities with Taylor interval approximations. In: NASA Formal Methods, *Lecture Notes in Computer Science*, vol. 7871, pp. 383–397. Springer (2013)
49. Strang, G.: The Fudamental Theorem of Linear Algebra. The American Mathematical Monthly **100**(9), 848–855 (1993)
50. Strang, G.: Introduction to Linear Algebra, 4th edn. Wellesley-Cambridge Press (2009)

51. Thiemann, R.: Implementing field extensions of the form $\mathbb{Q}[\sqrt{b}]$. Archive of Formal Proofs (2014). http://afp.sf.net/entries/Real_Impl.shtml, Formal proof development
52. Thiemann, R., Yamada, A.: Matrices, Jordan Normal Forms, and Spectral Radius Theory. Archive of Formal Proofs (2015). http://afp.sf.net/entries/Jordan_Normal_Form.shtml, Formal proof development
53. Thiemann, R., Yamada, A.: Algebraic Numbers in Isabelle/HOL (2016). Accepted for presentation in ITP 2016
54. Wenzel, M.: Isabelle/Isar — a versatile environment for human-readable formal proof documents. Ph.D. thesis, Technische Universität München (2002). https://mediatum.ub.tum.de/doc/601724/601724.pdf