REGULAR-MT: A Formal Proof of the Computation of Hermite Normal Form In a General Setting[⋆]

Jose Divasón and Jesús Aransay

Universidad de La Rioja, Logroño, La Rioja, Spain
{jose.divason,jesus-maria.aransay}@unirioja.es

Abstract. In this work, we present a formal proof of an algorithm to compute the Hermite normal form of a matrix based on our existing framework for the formalisation, execution, and refinement of linear algebra algorithms in Isabelle/HOL. The Hermite normal form is a well-known canonical matrix analogue of reduced echelon form of matrices over fields, but involving matrices over more general rings, such as Bézout domains. We prove the correctness of this algorithm and formalise the uniqueness of the Hermite normal form of a matrix. The succinctness and clarity of the formalisation validate the usability of the framework.

Keywords: Hermite normal form, Bézout domains, parametrised algorithms, Linear Algebra, HOL

1 Introduction

Computer algebra systems are neither perfect nor error-free, and sometimes they return erroneous calculations [18]. Proof assistants, such as Isabelle [36] and Coq [15] allow users to formalise mathematical results, that is, to give a formal proof which is mechanically checked by a computer. Two examples of the success of proof assistants are the formalisation of the four colour theorem by Gonthier [20] and the formal proof of Gödel's incompleteness theorems by Paulson [39]. They are also used in software [33] and hardware verification [28]. Normally, there exists a gap between the performance of a verified program obtained from a proof assistant and a non-verified one. However, research in this area is filling this gap to obtain efficient and verified programs which can be used for real applications and not just restricted to toy examples [6]. Linear algebra algorithms are widely used in mathematics and computer software due to their numerous applications in various fields, such as modern 3D graphics, search engines and modern compression algorithms. In this paper, we present a formalisation of an algorithm to compute the Hermite normal form of a matrix.

The Hermite normal form is a well-known canonical matrix that plays an important role in different fields. It can be used to solve algorithmic problems in lattices [21], cryptography [45], loop optimisation techniques [40], solution of systems of linear diophantine equations [10], and integer programming [25], among other applications.

[⋆] This work has been supported by the projects MTM2014-54151-P and MTM2017-88804-P from Ministerio de Economía y Competitividad (Gobierno de España).

The paper is structured as follows. We present a brief introduction to the Isabelle interactive theorem prover in Section 2. In Section 3 we describe the main features of our existing framework, where algorithms over matrices can be formalised, executed, refined, and coupled with their mathematical meaning as well as we introduce some benchmarks and improvements that we have carried out in this work. As a use case, we present in Section 4 the main contribution of this paper, *i.e.*, a formal proof of an algorithm to compute the Hermite normal form of a matrix in a general setting and the uniqueness of Hermite normal forms. A study of related and further work is given in Section 5. Finally, we show the conclusions in Section 6.

2 A brief introduction to Isabelle/HOL

Isabelle is a generic interactive proof assistant in which several logics are implemented. The most used of them is HOL (Isabelle/HOL), a version of classical higher-order logic similar to the one of the HOL System [5]. The Isabelle/HOL type system resembles that of functional programming languages such as Haskell [23]. There are base types (such as `bool`), function types representing total functions (*i.e.* ⇒), type constructors (such as `list`), and type variables (such as `'a` and `'b`).

For instance, `f :: 'a ⇒ 'b set` indicates that `f` is a function that maps an element of type `'a` to a set of elements of type `'b`. Isabelle/HOL also introduces type classes in a Haskell-like manner. A type class is just a group of types with a common interface: all types in that class must provide the functions in the interface. A type class not only provides an interface, but also allows to encode properties of the types. A type `'a` being in a class `B` is written `'a :: B`. Since our formalisation is based on Isabelle/HOL, throughout the paper we present the theorems and definitions following its syntax. Isabelle's keywords are written in **bold**. For a complete introduction on this proof assistant we refer the reader to [37]. All our quoted developments are publicly available in the Archive of Formal Proofs (AFP) [1, 17], which is a refereed repository of formal proof libraries developed in Isabelle.

3 Framework

In the last few years, we have completed several linear algebra developments in Isabelle/HOL [6–8]. They are based on the HOL Analysis (*HA*) library where a vector (type `vec`) is encoded as a function over a finite type (following the seminal work by Harrison [22]) and, consequently, a matrix is represented as a vector of vectors. More concretely, we developed a framework where linear algebra algorithms can be formalised, executed, refined, and coupled with their mathematical meaning. This framework includes, for instance, connection between linear maps and matrices, necessary generalisations of the HA library, formalisation of elementary operations of matrices, formalisation of the fundamental theorem of linear algebra, symbolic execution, a full library of algebraic structures (Bézout domains, principal ideal domains, GCD rings, *etc.*), connection with the Cayley-Hamilton theorem and so on.

As use cases, we implemented the Gauss-Jordan algorithm, the QR decomposition and the echelon form algorithm. All of them are available in the AFP [1]. Thiemann and Yamada used the framework in their formalisation of Jordan normal forms [44] (it is worth noting that they use a different representation of vectors from the one we use, by means of functions over the natural numbers with explicit dimensions associated to them). Also Li and Paulson reused our generalisations of the HA library in their work on real algebraic numbers [32]. Some parts have also been moved to the standard Isabelle/HOL library.

3.1 Main parts

The main parts of this framework are as follows:

1. Formalisation of elementary operations of matrices. We have defined them in Isabelle/HOL using the `vec` representation. For instance, we show here the definition of interchanging two rows of a matrix:

   definition *interchange_rows A a b = (χ i j. if i = a then A $ b $ j else if i = b then A $ a $ j else A $ i $ j)*

   In the above definition, χ denotes the morphism from functions to type `vec` and `$` is the access operator for `vec`.
2. Refinements from `vec` to executable representations (details can be found in [6]). We developed a *natural* refinement, from `vec` to functions over a finite type. We also developed a refinement to immutable arrays, or `iarray` to improve performance. An example of code lemma to transform from `vec` to `iarray` follows:

   lemma *[code-unfold]: matrix_to_iarray (interchange_rows A i j) = interchange_rows_iarray (matrix_to_iarray A) (to_nat i) (to_nat j)*

3. Serialisations to obtain better performance when generating code to functional programming languages (SML and Haskell). We have used two kinds of serialisations:
   - Immutable arrays (the efficient type used to represent vectors and matrices).
   - $\mathbb{Z}_2$, $\mathbb{Q}$, and $\mathbb{R}$ numbers (the types of the coefficients of the matrices).

   The latter ones are trivial. The first one in SML [4] was a part of the library. Regarding the serialisation of arrays in Haskell [2], we have serialised the `iarray` Isabelle/HOL datatype to the *Data.Array.IArray.array* (or shorter, *IArray.array*) constructor present in the standard Haskell library. Let us note that arrays are a natural way to represent *dense matrices*, which are the ones we are focusing in (we compute normal forms of matrices by means of elementary row operations). There exist also sparse representations of matrices in Isabelle/HOL by means of lists (see for instance the work by Obua and Nipkow [38]).

In the next subsection, we present some computational experiments that we completed and that justify our choice of immutable arrays for generating code of linear algebra algorithms from Isabelle/HOL specifications.

3.2 Performance

There exist different implementations of immutable arrays in Haskell, such as *IArrays* (*Data.Array.IArray.array*) or *UArrays* (*Data.Array.Unboxed.array*). In the case of the code generated from our Isabelle/HOL developments, we have empirically tested that *IArray.array* performs slightly better than *Unboxed.array*. As an example, the computation of the determinant of a 1500×1500 $\mathbb{Z}_2$ matrix by means of the code generated to Haskell from our verified Gauss-Jordan algorithm took $6.09s$ using *IArray.array* and $6.37s$ using *Unboxed.array*.[1] A more specific Haskell module for immutable arrays is *Data.Array* (where the *Data.Array.array* constructor is involved). As in the case of unboxed immutable arrays, the use of *Data.Array.array* does not imply an empirical advantage in terms of performance in our particular setup with respect to *Data.Array.IArray.array*.

We also perform some benchmarks in order to compare the performance of `vec` implemented as functions over finite domains, as immutable arrays, and also as lists (using an existing AFP entry about an implementation of matrices as lists of lists [42]). To do that, we define recursive functions (one for each representation: *function over finite domains*, *immutable arrays*, and *lists*) which take a rational matrix A as their input, and in each iteration interchange two rows of $A + A$.

Benchmarks are carried out for $10n \times 10n$ identity matrices, n being the number of iterations. Concretely, we execute the previous functions in two cases: applied to a 50×50 identity matrix with $n = 5$ and to the 100×100 identity matrix with $n = 10$. The algorithm is applied to identity matrices to minimize arithmetic time consumption. Table 1 shows the performance obtained when executing them within Isabelle/HOL (by means of the simplifier, with fully symbolic evaluation and highest confidence), and exporting code to SML by means of the command `code` to obtain better performance (in the second case, part of the code generation process is not verified, and needs to be trusted). Results show that the case $n = 5$ is usable in practice with any of the three representations. However, for bigger matrices, functions over finite domains become too slow. Immutable arrays outperform functions and lists in any case when exporting code, as expected. It is worth noting that inside Isabelle/HOL (but not when code is exported), `iarray` is just a wrapper of the type `list`. Thus, in the quest for performance, immutable arrays yield reasonable performance.

Focusing on our linear algebra algorithms, the performance obtained using functions over finite domains makes algorithms based on this representation unusable in practice. For instance, the computation of the Gauss-Jordan algorithm over 15×15 matrices is rather slow (several minutes). Using immutable arrays the computation is done immediately.

		functions	**iarray**	**list**
$n = 5$	**simp**	241.158s	-	20.860s
	code	0.639s	0.159s	0.971s
$n = 10$	**code**	827.673s	0.881s	1.824s

Table 1. Comparative among matrix representations.

The benchmarks and the execution examples presented throughout the paper have been carried out in a laptop with an Intel® Core$^{\text{TM}}$ i7-4810MQ processor with 16GiB

[1] The Isabelle file that serialises `iarray` to *UArrays* is available from our website [16].

of RAM and Ubuntu GNU/Linux 16.04. The code developed to carry out the benchmarks can be obtained online [16] and works for Isabelle 2017.

Mutable arrays (and imperative programming) should also be a good choice. Nevertheless, we compared the performance of using immutable arrays and mutable arrays in our formalisation of the Gauss-Jordan algorithm and obtained similar results [6].

4 A Formalisation of the Hermite Normal Form of a matrix

The Hermite normal form is commonly defined for integer matrices, but it also exists for more general matrices. Following a similar approach as the one that we followed in the formalisation of an algorithm to compute the echelon form of a matrix [8], we implemented an algorithm to compute the Hermite normal form for matrices whose elements belong to a Bézout domain. Execution is guaranteed for matrices over any Euclidean domain, since there always exists an executable operation for computing Bézout coefficients. This executable operation over Euclidean domains is already implemented in the Isabelle/HOL standard library by Eberl. One could also execute the algorithm with matrices over a Bézout domain, as long as an executable operation to compute Bézout coefficients is provided.

4.1 Definition of Hermite normal form

Our formalisation of the Hermite normal form is built from many pieces. Essentially we need matrices and polynomials from the standard library and from our framework:

- Generalisations of the HA library.
- Elementary operations over matrices, executability of the `vec` representation, serialisations to obtain efficient code.
- An algorithm to compute the echelon form of a matrix.
- Ring theory (some fragments were already present in the standard library).

Let us stress that there is no unique definition of Hermite normal form in the literature. For instance, some authors, like Newman [35], restrict their definitions to the case of square nonsingular matrices (that is, invertible matrices). Other authors, like Cohen [13], just work with integer matrices. Furthermore, given a matrix A its Hermite normal form H can be defined to be upper triangular [43] or lower triangular [35]. In addition, the transformation from A to H can be made by means of elementary row operations [35] or elementary column operations [13]. In this formalisation, we work as generally as possible, so we do not impose restrictions in the input matrix (coefficients must belong to a Bézout domain and both square and non-square matrices are accepted).

In our algorithm the transformation to the Hermite normal form is carried out by means of *elementary row operations*, obtaining H as an upper triangular matrix. This design decision will allow us to reuse our previous work. Moreover, any algorithm or theorem using an alternative definition of Hermite normal form (for example, in terms of column operations and/or lower triangularity) can be moulded into the form of Definition 4.

Firstly, we have to define the concepts of *complete set of nonassociates* and *complete set of residues modulo μ*. Let $\mathcal{R}$ be a commutative ring with unit.

Definition 1 (Complete set of nonassociates). *An element $a \in \mathcal{R}$ is said to be an* associate *of an element $b \in \mathcal{R}$ if there exists an invertible element $u \in \mathcal{R}$ such that $a = ub$. This is an equivalence relation over $\mathcal{R}$. A set of elements of $\mathcal{R}$, one from each equivalence class, is said to be a* complete set of nonassociates.

Definition 2 (Complete set of residues). *Let μ be any nonzero element of $\mathcal{R}$. Let $a, b \in \mathcal{R}$; a is* congruent *to b modulo μ if μ divides $a - b$. This is an equivalence relation over $\mathcal{R}$. A set of elements of $\mathcal{R}$, one from each equivalence class, is said to be a* complete set of residues modulo μ *(or shorter, a* complete set of residues of μ*).*

Let us start introducing the Isabelle/HOL implementation of associated (due to Eberl) and congruent elements (this and the following definitions are available from file *Hermite.thy* of our development [17]). In the definitions, `x dvd y` means that the element `x` divides the element `y`:

```
definition associated x y ⟷ x dvd y ∧ y dvd x
definition cong a b u = (u dvd (a - b))
```

We easily connect Eberl's definition of associated elements with Definition 1 and show they are equivalent.

```
lemma associates a b = (∃u∈Units. a = u * b)
```

Next, we define the corresponding relations of associates and congruence introduced by the definitions. We define the relations by means of sets. Two elements (a, b) belong to the set if they are related. Hence:

```
definition associated_rel = {(a, b). associated a b}
definition congruent_rel u = {(a, b). cong a b u}
```

We prove both of them to be reflexive, transitive, and symmetric (*i.e.*, they are equivalence relations over `UNIV`, where `UNIV` represents the set of all elements of the ring).

```
lemma equiv UNIV associated_rel
lemma equiv UNIV (congruent_rel u)
```

From the definitions of associated and congruent elements, we introduce the *complete set of nonassociates* and *complete sets of residues modulo an element*. Authors usually avoid these definitions imposing additional conditions to the Hermite normal form. For instance, in the particular case of integers, the residues r modulo μ are usually chosen such that $0 \le r < \mu$ (see [13]), but $-\mu < r \le 0$ (see [10]) and $-\frac{\mu}{2} < r \le \frac{\mu}{2}$ (see [3]) are also valid choices. Every possibility fits selecting a complete set of nonassociates and complete sets of residues.

A function f is an *associates function* if for all $a \in \mathcal{R}$, then a and $f(a)$ are associated. In order to obtain a complete set of nonassociates, we impose the elements belonging to the range of f to be pairwise nonassociates. Hence, a set S will be a complete set of nonassociates if there exists an associates function f whose range is S.

```
definition ass_function f = ((∀a. associated a (f a)) ∧
  pairwise (λa b. ¬ associated a b) (range f))
definition Complete_set_non_associates S =
  (∃f. ass_function f ∧ range f = S)
```

Such definitions satisfy the following properties:

```
lemma assumes ass_function f
shows Complete_set_non_associates (range f)
lemma assumes Complete_set_non_associates S
and x ∈ S and y ∈ S and x ≠ y shows ¬ associated x y
```

A function f is a *residues function* if, given $u \in \mathcal{R}$, the following conditions hold:

1. For all $a, b \in \mathcal{R}$, a and b are congruent modulo u if and only if $f\ u\ a = f\ u\ b$.
2. The elements which belong to the range of f are pairwise noncongruent modulo u.
3. For all $a \in \mathcal{R}$, $f\ u\ a$ and a are congruent modulo u

```
definition res_function f =
(∀u. (∀a b. cong a b u ⟷ f u a = f u b)
∧ pairwise (λa b. ¬ cong a b u) (range (f u))
∧ (∀a. cong (f u a) a u))
```

Essentially, the residue function picks out an element for each residue class. From the latter condition it follows that the elements (such as a) above a leading entry (such as u) can be converted to $f\ u\ a$ by elementary operations, since there exists $k \in \mathcal{R}$ such that $f\ u\ a = a + ku$. There exists a complete set of residues for each element $u \in \mathcal{R}$. Thus, g models a complete sets of residues if there exists a residues function f such that each set $g\ u$ is exactly the range of $f\ u$:

```
definition Complete_set_residues g =
(∃f. res_function f ∧ (∀u. g u = range (f u)))
```

The function satisfies the expected properties:

```
lemma assumes f: res_function f
shows Complete_set_residues (λu. range (f u))
lemma assumes Complete_set_residues g
and x ∈ g b and y ∈ g b and x ≠ y shows ¬ cong x y b
```

We can provide (executable) associates and residues functions involving elements over Euclidean domains:

```
definition ass_function_euclidean p = normalize p
definition res_function_euclidean b n= (if b=0 then n else n mod b)
```

In the above definitions, `normalize` specifies a canonical representant for each equivalence class in the Euclidean domain. For instance, in the case of the integers, `normalize` corresponds to the absolute value. The functions are proven to be associates and residues functions respectively:

```
lemma ass_function ass_function_euclidean
lemma res_function res_function_euclidean
```

We could also provide other different instances of associates and residues functions. For instance, the minus absolute value can be used as an associates function for integer elements:

```
lemma ass_function (λn::int. -abs n)
lemma range (λn::int. -abs n) = {x. x ≤ 0}
```

With the previous ingredients we can now introduce the definition of the Hermite normal form.

Definition 3 (Echelon form). *A matrix $H \in M_{m\times n}(\mathcal{R})$ is said to be in echelon form if:*

1. *All rows consisting only of 0's appear at the bottom of the matrix.*
2. *For any two consecutive nonzero rows, the leading entry of the lower row is to the right of the leading entry of the upper row.*

Definition 4 (Hermite normal form). *Given a complete set of nonassociates S and complete sets of residues G, a matrix $H \in M_{m\times n}(\mathcal{R})$ is said to be in Hermite normal form if:*

1. *H is in echelon form.*
2. *The leading entry of every nonzero row belongs to S.*
3. *Let h be the leading entry of a nonzero row. Then each element above h belongs to $G\ h$.*

Our Isabelle/HOL implementation of the definition is parametrised by a matrix A and two functions, `associates` and `residues`, which are demanded to be associates and residues functions respectively. The operator `LEAST n. P n` returns the least element `n` that satisfies a property `P`, in our case the least index `n` such that `A$i$n≠0`.

```
definition Hermite associates residues A =
(Complete_set_non_associates associates
∧ Complete_set_residues residues ∧ echelon_form A
∧ (∀i. ¬ is_zero_row i A ⟶
      A $ i $ (LEAST n. A $ i $ n ≠ 0) ∈ associates)
∧ (∀i. ¬ is_zero_row i A ⟶
       (∀j. j<i ⟶ A$j$(LEAST n. A $ i $ n ≠ 0)
        ∈ residues (A$i$(LEAST n. A$i$n ≠ 0)))))
```

Definition 5 (Hermite normal form of a matrix). *A matrix $H \in M_{m\times n}(\mathcal{R})$ is the Hermite normal form of a matrix $A \in M_{m\times n}(\mathcal{R})$ if:*

1. *There exists an invertible matrix P such that $A = PH$.*
2. *H is in Hermite normal form.*

4.2 An algorithm to compute the Hermite normal form of a matrix

Any matrix over a Bézout domain can be transformed by means of elementary operations to its Hermite normal form. A schema of the computation of the Hermite normal form is presented in Algorithm 1. There exist more efficient (in both computational cost and space consumption) algorithms to compute the Hermite normal form of a matrix. Normally they are restricted to specific domains, such as polynomial matrices [26].

Algorithm 1: An algorithm to compute the Hermite normal form of a matrix A

```
Input: $A \in M_{m\times n}(\mathcal{B})$ and complete sets of nonassociates and residues.
Output: A matrix $H$ such that $\exists P.\ A = PH$, where $P \in M_{m\times m}(\mathcal{B})$ is invertible and
        $H \in M_{m\times n}(\mathcal{B})$ is in Hermite normal form with respect to the given complete
        sets of nonassociates and residues.
1 Transform the matrix $A$ to its corresponding echelon form;
2 Transform each row such that its leading entry belongs to the complete set of
    nonassociates, multiplying each row by an appropriate constant;
3 Transform the elements above each leading entry, i.e., such elements must belong to the
    corresponding complete set of residues with respect to the leading entry. This is done by
    adding to each row above the leading entry, the row of the leading entry multiplied by a
    constant (that is, the transformation is carried out by means of elementary operations).
```

We have implemented the Hermite algorithm in Isabelle/HOL iterating over rows. That is, we have defined an operation that carries out the transformations over one row and then we have defined the Hermite algorithm folding such an operation over all rows. Our Hermite algorithm relies on our previous version of the echelon form algorithm [8]. The algorithm is parametrised by three functions:[2]

- A function that computes Bézout's identity of two elements (required for the echelon form).
- An associates function whose range is a complete set of nonassociates.
- A residues function whose range consists of complete sets of residues.

The Hermite algorithm must be parametrised with the functions that satisfy the required properties presented above. The proof of correctness of the algorithm will assume that such functions are really Bézout, associates, and residues functions respectively. These requirements are expressed by means of premises.

[2] Neither records nor locales [9] are used for this task, although they are a valid alternative.

The following Isabelle functions reproduce the steps in Algorithm 1. Step 1 corresponds with the function *echelon_form_of* of our previous work. Step 3 is performed by means of *Hermite_reduce_above* starting from the proper index (one of its parameters is the residues function). We use a primitive recursive definition over the representation of the row-indexes as natural numbers.

```
primrec Hermite_reduce_above A 0 i j res  = A
     | Hermite_reduce_above A (Suc n) i j res  =
(let i'=((from_nat n)::'rows); Aij = A $ i $ j;  Ai'j = A$i'$j;
      k = (((res Aij (Ai'j))-(Ai'j)) div Aij) in
Hermite_reduce_above (row_add A i' i k) n i j res)
```

This function is reused in *Hermite_of_row_i*, which performs Step 2 (it also has both the associates and the residues functions as parameters).

```
definition Hermite_of_row_i ass res A i =
 (if is_zero_row i A then A else
   let j = (LEAST n. A $ i $ n ≠ 0); Aij= (A $ i $ j);
   A' = mult_row A i ((ass Aij) div Aij)
 in Hermite_reduce_above A' (to_nat i) i j res)
```

The function *Hermite_of_upt_row_i* iterates the process up to a row i.

```
definition Hermite_of_upt_row_i A i ass res =
  foldl (Hermite_of_row_i ass res) A (map from_nat [0..<i])
```

Finally, *Hermite_of* takes echelon form as starting point and applies the function *Hermite_of_upt_row_i* to its rows:

```
definition Hermite_of A ass res bezout = (let A'= echelon_form_of A
bezout in Hermite_of_upt_row_i A' (nrows A) ass res)
```

The soundness of the algorithm can be split into four parts:

1. The output matrix is in echelon form.
2. Each leading entry belongs to the complete set of nonassociates.
3. Each element above a leading entry belongs to the corresponding complete set of residues.
4. The algorithm is carried out by means of elementary row operations (therefore, the *output* matrix is the Hermite normal form of the input matrix).

Part 1 takes advantage of our previous proof about echelon forms [8], and requires proving that *Hermite_of_upt_row_i* preserves echelon forms. This property follows from the definition of *Hermite_reduce_above*, since it performs elementary row operations only above the leading coefficients of each row. Thus, it does not alter any of the properties of the echelon form. Part 2 easily follows from the definition *Hermite_of_row_i*. The proof of Part 3 is based on the definition of *Hermite_reduce_above*. The proof is more intricate, since we have to prove the result for one row, and then apply inductively the result to the rest of rows (proving that the previous ones are preserved in each iteration). The crucial lemma for this part states that the property holds when the algorithm is iteratively applied up to the $k-th$ row:

lemma defines *n=(LEAST n. A $ i $ n ≠ 0)*
assumes *¬ is_zero_row i A* **and** *echelon_form A* **and** *ass_function ass*
and *res_function res* **and** *to_nat i < k* **and** *k ≤ nrows A* **and** *j < i*
shows *(Hermite_of_upt_row_i A k ass res) $ j $ n*
   ∈ range (res (Hermite_of_upt_row_i A k ass res $ i $ n))

Finally, Part 4 is established by proving that the required steps to compute the Hermite normal form can be expressed as invertible matrices (here we also reuse results of our previous developments), and therefore are equivalent to elementary operations. We refer the interested reader to the file *Hermite.thy* of the development for the full-detailed proofs and statements. In a modest 1400 code lines we obtain the final theorem:

theorem assumes *ass_function ass*
and *res_function res* **and** *is_bezout_ext bezout*
shows *∃P. invertible P ∧ (Hermite_of A ass res bezout) = P ** A*
∧ Hermite (range ass) (λc. range (res c)) (Hermite_of A ass res bezout)

In *ca.* 150 Isabelle/HOL code lines, we refine the algorithm to immutable arrays and generate its SML and Haskell versions (see file *Hermite_IArrays.thy*).

4.3 Uniqueness

Theorem 1. *Fixing a complete set of nonassociates and complete sets of residues, if $A \in M_{n\times n}(\mathcal{R})$ is a nonsingular matrix, then its Hermite normal form is unique.*

Let us note that the Hermite normal form of an invertible matrix is the identity matrix when the standard associates and residues functions over euclidean domains are chosen in the algorithm, but this does not hold in general. In order to prove Theorem 1, we follow the proof by Newman [35, Theorem II.3]. Where Newman considers the Hermite normal form as a lower triangular matrix we consider it upper triangular.

lemma assumes *A = P ** H* **and** *A = Q ** K* **and** *invertible A*
and *invertible P* **and** *invertible Q*
and *Hermite associates residues H*
and *Hermite associates residues K* **shows** *H = K*

The original proof comprises 28 lines [35, Th. II.3]. The argument proceeds as follows: let us suppose that, for a given nonsingular matrix A, there are two different upper triangular Hermite forms (*wrt* the same sets of associates and residues), H and K. Then, there exists a unit matrix U such that $H = UK$. U must also be upper triangular. Its diagonal elements are 1, since $h_{ii} = u_{ii}k_{ii}$ with both h_{ii}, k_{ii} in the same set of nonassociates. The remaining elements of the matrix must be equal to 0. Let $s \in \{0 \ldots n-1\}$ (any valid row). Since the matrix is upper triangular, we consider the element $u_{s,s+j}$ with $j \in \{1 \ldots (\text{ncols}\, A - s)\}$ (*i.e.*, any element above the diagonal). We apply *total induction* in j, and therefore we assume that $u_{s,s+1}, \ldots, u_{s,s+(j-1)}$ are equal to 0. Hence:

$$
\begin{aligned}
h_{s,s+j} &= \sum_{t=0}^{nrows\,A} u_{st} k_{t(s+j)} && (1)\\
&= \sum_{t=s}^{s+j} u_{st} k_{t(s+j)} && (2)\\
&= u_{ss} k_{s,s+j} + u_{s,s+1} k_{s+1,s+j} + \cdots + u_{s,s+j} k_{s+j,s+j} && (3)\\
&= u_{ss} k_{s,s+j} + u_{s,s+j} k_{s+j,s+j} && (4)\\
&= k_{s,s+j} + u_{s,s+j} k_{s+j,s+j} && (5)
\end{aligned}
$$

Step 2 follows from K being upper triangular; step 4 follows from the induction hypothesis; step 5 follows from the first part of the proof ($u_{ii} = 1$). Therefore, $h_{s,s+j} \equiv k_{s,s+j}$ mod $k_{s+j,s+j}$, from where it follows that $h_{s,s+j} \equiv k_{s,s+j}$, since both elements belong to the same complete set of residues of $k_{s+j,s+j}$ (and hence $u_{s,s+j} = 0$).

This inductive reasoning, which in the original proof took 18 lines, required 88 lines in our formalisation. The proof itself is not particularly intricate, but it demands a correct manipulation of the indexes. The complete proof took 150 lines, thanks to the strong reuse of previous results already available in the framework. It firmly follows the book proof line by line.

4.4 Examples of execution

We provide two examples of execution of our formalised algorithm. Both use the *standard* associates and residues functions, which are defined for Euclidean domains. Let us choose a rectangular random integer matrix A and a polynomial matrix B.

$$
A = \begin{bmatrix} 37 & 9 & 10 & 28 & 40 & 23 & 59 & 25 & 73 & 79 \\ 5 & 96 & 93 & 7 & 71 & 44 & 63 & 90 & 27 & 89 \\ 70 & 65 & 36 & 69 & 2 & 81 & 14 & 30 & 92 & 60 \\ 16 & 98 & 100 & 50 & 64 & 21 & 39 & 95 & 80 & 34 \end{bmatrix}, B = \begin{bmatrix} 5x^2 + 4x + 3 & x - 2 \\ 2x^2 - 1 & x^3 + 4x^2 + x \end{bmatrix}
$$

Their Hermite normal forms are computed in Isabelle/HOL similarly:

```
value [code] matrix_to_list_of_list (Hermite_of M
ass_function_euclidean res_function_euclidean euclid_ext2)
```

Where `M` is a matrix in Isabelle/HOL that corresponds to A or B, depending on the example we are executing. The function `Hermite_of` has four parameters: the input matrix M, the *standard* associates function for Euclidean domains (`ass_function_euclidean`), the *standard* residues function for Euclidean domains (`res_function_euclidean`), and the function which computes Bézout coefficients in Euclidean domains (`euclid_ext2`), which is required by the echelon form algorithm. Let us note that the type inference will determine which version of the associates and residues functions must be executed, depending on the type of the input matrix. The function *matrix_to_list_of_list* eases outputting matrices. The obtained results follow:

$$\begin{bmatrix} 1 & 0 & 0 & 2126849 & -2040340 & -1544323 & -3517370 & -665650 & 1303207 & -5664981 \\ 0 & 1 & 0 & 3330071 & -3194626 & -2417993 & -5507258 & -1042230 & 2040466 & -8869838 \\ 0 & 0 & 1 & 1681610 & -1613209 & -1221033 & -2781035 & -526300 & 1030392 & -4479062 \\ 0 & 0 & 0 & 3802428 & -3647768 & -2760977 & -6288437 & -1190065 & 2329900 & -10127986 \end{bmatrix}$$

$$\begin{bmatrix} 1 & -\frac{44}{89} + \frac{31}{89}x - \frac{68}{89}x^2 + \frac{137}{89}x^3 + \frac{40}{89}x^4 \\ 0 & -\frac{2}{5} + \frac{4}{5}x + 4x^2 + \frac{22}{5}x^3 + \frac{24}{5}x^4 + x^5 \end{bmatrix}$$

The results satisfy the expected properties (for instance, the polynomial matrix has monic polynomials as leading entries in each row). Let us also note the growth on the size of the elements. Both matrices are computed instantly. Such examples can also be executed inside the logic, but they take 30 and 8 minutes respectively.

The performance of the algorithm is highly dependent on several factors. Some of them follow from our design choices, such as the selection of associates and residues functions, and the function to compute the Bézout identity. Some others depend on the system configuration, such as the serialisations employed. Finally, the chosen algorithm to compute the Hermite normal form itself can be extremely space consuming (there exist versions that bound the size of the intermediate entries computed [27]). With our particular version, the time to compute the Hermite normal form of a 20×20 integer matrix with random entries between 0 and 100 making use of the *standard* associates, residues, and Bézout functions is negligible using the refinement to *iarray*. The resulting matrix has elements with more than 50 digits. The same happens with a 25×25 integer matrix. Memory issues appear with higher dimensions and greater elements.

5 Related and further work

Linear algebra has been formalised in many theorem provers: Isabelle/HOL [44], Coq [12], Mizar [41], HOL Light [22], PVS [34], and ACL2 [19] are just a few examples of it. On the contrary, the verification and implementation of linear algebra algorithms have not been so widely explored, especially involving matrices over rings. The most similar works have been carried out in Coq. It is worth citing the CoqEAL development [12], which contains several linear algebra algorithms formalised in Coq, such as the Sasaki-Murao [14] algorithm for computing determinants of matrices over rings. The closest work to ours is the one done by Cano *et al.* [11] also in Coq, which presents a formalisation of the Smith normal form (SNF) of a matrix. Their formalisation is restricted to explicit division rings, such as constructive Bézout domains, whereas in our case we can work with more abstracts structures where the existence of divisions and greatest common divisors are known, but maybe not how to compute them. In any case, the SNF algorithm is distinct from the Hermite normal form. SNF requires both row and column operations and the result is a diagonal matrix. The Hermite normal form sometimes can be view as a previous step, but is not required to compute the SNF. The computation of the echelon form of integer matrices has been recently formalised in ACL2 by Lambán *et al.* [30] as an application of abstract single threaded objects. A formal proof of the SNF would be desirable in Isabelle/HOL. Most of the algorithms to compute the SNF of a matrix are based on submatrices [43]. Unfortunately, submatrices are a delicate issue in the HA library: Since Isabelle does not feature dependent types, we cannot use the size of the matrix in the definition of submatrix. Thiemann and Yamada already faced this problem when formalising Jordan normal forms of matrices [44], a kind of forms whose construction is done by means of block matrices. As a solution, they propose a new matrix representation which is indeed an abstraction of the HA representation, but flexible for dimensions. We aim to formalise the Smith normal form in Isabelle using such a representation, also connecting it to the HA library and our framework by means of the lifting and transfer package [24] and the new addition of local type definitions when necessary [29]. This would also allow us to implement some decision procedures based on linear algebra methods, such as the decision algorithm proposed by Li *et al* [31].

6 Conclusions

We have presented a formalisation of the Hermite normal form of a matrix that reuses our previous developments. The Hermite normal form of a matrix is a well-known canonical matrix over rings. We have not only proved the correctness of the algorithm, but we have also refined it to immutable arrays and we have formalised its uniqueness as well. As far as we know, this is the first formalisation of the Hermite normal form in any theorem prover, even only considering the case of integer matrices.

The formalisation could be seen as a proof pearl because of two reasons: we have formalised a non-trivial and well-known linear algebra algorithm in a modest number of lines (*ca.* 2300, to be compared with more than 10000 that we needed for similar results with the Gauss-Jordan algorithm) thanks to a strong reuse of the infrastructure presented in Section 3; and we have focused on obtaining the most general version by means of a parametrised algorithm. The formalisation has been carried out involving matrices over Bézout domains and it is not restricted to the common case of integer matrices. Furthermore, the algorithm has been parametrised by functions so that it can compute every definition of the Hermite normal form in the literature.

References

1. Archive of Formal Proofs. `http://afp.sourceforge.net/`.
2. Immutable non-strict arrays in Haskell. `http://hackage.haskell.org/package/array-0.5.2.0/docs/Data-Array.html`.
3. Matlab documentation. Definition of Hermite Normal Form. `http://es.mathworks.com/help/symbolic/hermiteform.html#butzrp_-5`.
4. The Vector structure in SML. `http://sml-family.org/Basis/vector.html`.
5. HOL interactive theorem prover. `https://hol-theorem-prover.org/`, 2016.

6. J. Aransay and J. Divasón. Formalisation in higher-order logic and code generation to functional languages of the Gauss-Jordan algorithm. *Journal of Functional Programming*, 25, 2015.
7. J. Aransay and J. Divasón. A Formalisation in HOL of the Fundamental Theorem of Linear Algebra and Its Application to the Solution of the Least Squares Problem. *Journal of Automated Reasoning*, pages 1–27, 2016.
8. J. Aransay and J. Divasón. Formalisation of the computation of the echelon form of a matrix in Isabelle/HOL. *Formal Aspects of Computing*, 28(6):1005–1026, 2016.
9. C. Ballarin. Locales: A module system for mathematical theories. *Journal of Automated Reasoning*, 52(2):123–153, 2014.
10. G. H. Bradley. Algorithms for Hermite and Smith normal matrices and linear diophantine equations. *Mathematics of Computation*, 25(116):897–907, 1971.
11. G. Cano, C. Cohen, M. Dénès, A. Mörtberg, and V. Siles. Formalized Linear Algebra over Elementary Divisor Rings in Coq. *Logical Methods in Computer Science*, 12(2), 2016.
12. C. Cohen, M. Dénès, and A. Mörtberg. Refinements for Free! In *Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings*, pages 147–162, 2013.
13. H. Cohen. *A Course in Computational Algebraic Number Theory*. Springer-Verlag New York, Inc., New York, NY, USA, 1993.
14. T. Coquand, A. Mörtberg, and V. Siles. A formal proof of Sasaki-Murao algorithm. *Journal of Formalized Reasoning*, 5(1):27–36, 2012.
15. The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2018. Version 8.8.0. `http://coq.inria.fr`.
16. J. Divasón. Additional files to the Hermite normal form development, 2018. `http://www.unirioja.es/cu/jodivaso/Isabelle/Hermite/Hermite.html`.
17. J. Divasón and J. Aransay. Hermite normal form. *Archive of Formal Proofs*, July 2015.
18. A. J. Durán, M. Pérez, and J. L. Varona. The Misfortunes of a Trio of Mathematicians Using Computer Algebra Systems. Can We Trust in Them? *Notices of the AMS*, 61(10):1249 – 1252, 2014.
19. R. Gamboa, J. Cowles, and J. V. Baalen. Using ACL2 Arrays to Formalise Matrix Algebra. In *Fourth International Workshop on the ACL2 Theorem Prover and Its Applications*, 2003.
20. G. Gonthier. Formal proof – the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, 2008.
21. J. L. Hafner and K. S. McCurley. A rigorous subexponential algorithm for computation of class groups. *Journal of the American Mathematical Society*, 2(4):837–850, 1989.
22. J. Harrison. The HOL Light Theory of Euclidean Space. *Journal of Automated Reasoning*, 50(2):173 – 190, 2013.
23. The Haskell Programming Language. `http://www.haskell.org/`, 2016.
24. B. Huffman and O. Kuncar. Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL. In *Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings*, pages 131–146, 2013.
25. M. S. Hung and W. O. Rom. An application of the Hermite normal form in integer programming. *Linear Algebra and its Applications*, 140:163 – 179, 1990.
26. E. Kaltofen, M.S. Krishnamoorthy, and B.D. Saunders. Fast parallel computation of Hermite and Smith forms of polynomial matrices. *SIAM Journal on Algebraic Discrete Methods*, 8(4):683–690, 1987.
27. R. Kannan and A. Bachem. Polynomial algorithms for computing the Smith and Hermite normal forms of an integer matrix. *SIAM Journal on Computing*, 8(4):499–507, 1979.
28. G. Klein et al. seL4: formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles, SOSP 2009, Big Sky, Montana, USA*, pages 207–220, 2009.

29. O. Kunčar and A. Popescu. From types to sets by local type definitions in higher-order logic. In Jasmin Christian Blanchette and Stephan Merz, editors, *Interactive Theorem Proving*, pages 200–218, Cham, 2016. Springer International Publishing.
30. L. Lambán, F. J. Martín-Mateos, J. Rubio, and J.-L. Ruiz-Reina. Using abstract stobjs in ACL2 to compute matrix normal forms. In *Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings*, pages 354–370, 2017.
31. L. Li, H. Li, and Y. Liu. A decision algorithm for linear sentences on a PFM. *Annals of Pure and Applied Logic*, 59:273 – 286, 1993.
32. W. Li and L. C. Paulson. A modular, efficient formalisation of real algebraic numbers. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, FL, USA, January 20-22, 2016*, pages 66–75, 2016.
33. A. Lochbihler. Verifying a compiler for java threads. In A. D. Gordon, editor, *Programming Languages and Systems*, pages 427–447, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
34. A. Narkawicz, C. Muñoz, and A. Dutle. Formally-Verified Decision Procedures for Univariate Polynomial Computation Based on Sturm's and Tarski's Theorems. *Journal of Automated Reasoning*, 54(4):285–326, 2015.
35. M. Newman. *Integral matrices*. Pure and Applied Mathematics. Elsevier Science, 1972.
36. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.
37. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, 2018. Updated version of the book with the same title and authors.
38. S. Obua and T. Nipkow. Flyspeck II: the basic linear programs. *Ann. Math. Artif. Intell.*, 56(3-4):245–272, 2009.
39. L. C. Paulson. A Mechanised Proof of Gödel's Incompleteness Theorems Using Nominal Isabelle. *Journal of Automated Reasoning*, 55(1):1–37, 2015.
40. J. Ramanujam. Beyond unimodular transformations. *The Journal of Supercomputing*, 9(4):365–389, 1995.
41. P. Rudnicki, C. Schwarzweller, and A. Trybulec. Commutative Algebra in the Mizar System. *Journal of Symbolic Computation*, 32(1/2):143–169, 2001.
42. C. Sternagel and R. Thiemann. Executable matrix operations on matrices of arbitrary dimensions. *Archive of Formal Proofs*, June 2010.
43. A. Storjohann. *Algorithms for Matrix Canonical Forms*. PhD thesis, Swiss Federal Institute of Technology Zurich, 2000.
44. R. Thiemann and A. Yamada. Formalizing Jordan normal forms in Isabelle/HOL. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, FL, USA, January 20-22, 2016*, pages 88–99, 2016.
45. V. E. Tourloupis. Hermite normal forms and its cryptographic applications. Master's thesis, University of Wollongong, 2013.