A Formalization of Elementary Linear Algebra: Part I

David M. Russinoff
david@russinoff.com

This is the first installment of an exposition of an ACL2 formalization of elementary linear algebra, focusing on aspects of the subject that apply to matrices over an arbitrary commutative ring with identity, in anticipation of a future treatment of the characteristic polynomial of a matrix, which has entries in a polyniomial ring. The main contribution of this paper is a formal theory of the determinant, including its characterization as the unique alternating n-linear function of the rows of an $n \times n$ matrix, multiplicativity of the determinant, and the correctness of cofactor expansion.

1 Introduction

This is the first installment of an exposition of an ACL2 formalization of elementary linear algebra, covering the basic algebra of matrices and the theory of determinants. Part II [14], also included in this workshop, addresses row reduction and its application to matrix invertibility and simultaneous systems of linear equations. Additional topics to be covered in future installments include vector spaces, linear transformations, polynomials, eigenvectors. and diagonalization.

This ordering of topics departs from the typical syllabus of an introductory course in the subject. Most elementary linear algebra textbooks treat the solution of simultaneous linear equations in the first chapter, perhaps to reassure the student of the practical utility of the theory. Consequently, (since this process depends on the existence of a multiplicative inverse) the entries of a matrix are assumed at the outset to range over a field (often the real numbers) rather than a more general commutative ring. This assumption, however, is not required for the main results of matrix algebra or the properties of the determinant; in fact, there are numerous applications for which it does not hold [1]. Indeed, several chapters later, one finds that the theory of eigenvalues is based on the fundamental notion of the characteristic polynomial of a matrix over a field F, which is properly defined as the determinant of a matrix with entries in the polynomial ring $F[t]$. In most cases, this problem is simply ignored [5, 6, 10]. A rare exception is a comparatively rigorous treatment by Hoffman and Kunze [4], on which our formalization is partly based (and from which this author learned the subject as a college sophomore). In Chapter 5, (anticipating the introduction of the characteristic polynomial) they define the determinant of a matrix over an arbitrary commutative ring with unity and ask the reader to determine for himself which of the results of the preceding chapters, though stated and proved for matrices over a field, apply more generally to commutative rings.

Neither of these strategies will serve our purpose. Unconstrained by pedagogical considerations, we pursue a more principled development, separating those aspects of the theory that are valid for general commutative rings from those that depend on the existence of a multiplicative inverse. The former topics are treated in this paper; the latter in Part II. All supporting proof scripts reside in the shared ACL2 directory `books/projects/linear/`.

In Section 2, we introduce the notion of an abstract commutative ring with unity by means of an encapsulated set of constrained functions and associated theorems corresponding to the standard ring axioms. Section 3 covers the algebra of matrices and the transpose operator. The main contribution of this paper is a formal theory of determinants, based on the classical definition, which appeals to the properties

To appear in EPTCS.

of the symmetric group. This consideration was a factor in our broader plan of a formalization of algebra beginning with the theory of finite groups [11, 12, 13], on which the present work critically depends. In Section 4, we define the determinant and derive its main properties, including its uniqueness as an alternating n-linear function of the rows of an $n \times n$ matrix. Multiplicativity is derived as a consequence of this result, which is further exploited in Section 5 to establish the correctness of cofactor expansion and the properties of the classical adjoint. The proofs of uniqueness and its consequences illustrate the use of encapsulation and functional instantiation as a substitute for higher order logical reasoning in the first order logic of ACL2.

Previous work on matrix algebra within the ACL2 community includes a formalization by Gamboa et al. based on ACL2 arrays [2], another by Hendrix with matrices defined as simple list structures [3], and Kwan's proofs of correctness of several numerical algorithms [7, 8]. Our matrix representation scheme is essentially that of Hendrix (which was also adopted by Kwan), but since we require entries ranging over an abstract ring rather than the `acl2-number` type, ours is constructed independently. Only the first of these cited references provides a general definition of the determinant, with no proofs of its properties. A variety of linear algebra formalizations have been based on other theorem provers [9, 15, 16, 17, 18], but we are not aware of any that has produced the full list of results reported above.

2 Commutative Rings

In `ring.lisp`, the axioms of a commutative ring with unity are formalized by an encapsulation, partially displayed below:

```
(encapsulate (((rp *) => *)                    ;ring element recognizer
              ((r+ * *) => *) ((r* * *) => *) ;addition and multiplication
              ((r0) => *) ((r1) => *)          ;identities
              ((r- *) => *))                   ;additive inverse
  (local (defun rp (x) (rationalp x)))
  (local (defun r+ (x y) (+ x y)))
  (local (defun r* (x y) (* x y)))
  (local (defun r0 () 0))
  (local (defun r1 () 1))
  (local (defun r- (x) (- x)))
  ;; Closure:
  (defthm r+closed (implies (and (rp x) (rp y)) (rp (r+ x y))))
  (defthm r*closed (implies (and (rp x) (rp y)) (rp (r* x y))))
  ;; Commutativity:
  ...
}
```

This introduces six constrained functions: `rp` is a predicate that recognizes an element of the ring; `r+` and `r*` are the binary addition and multiplication operations; the constants `(r0)` and `(r1)` are the identity elements of these operations, respectively; and `r-` is the unary addition inverse. Note that these functions are locally defined to be the corresponding functions pertaining to the rational numbers (an arbitrary choice—the recognizer `(integerp x)` would have worked just as well as `(rationalp x)`). The exported theorems (mostly omitted above) are the usual ring axioms: closure, commutativity, and associativity of both operations; properties of the identities and the additive inverse; and the distributive law. Informally, we shall refer to the ring `R` that is characterized by these axioms, and elements of `R` are sometimes called *scalars*. When our intention is clear, we may abbreviate `(r0)` and `(r1)` as 0 and 1, respectively.

The file also contains some trivially derived variants of the axioms, along with definitions of several functions pertaining to lists of ring elements and proofs of their basic properties:

- `rlistp` is a predicate that recognizes a vector, i.e., a proper list of scalars, which we call an *rlist*;
- `rlistnp` recognizes an rlist of a specified length;
- `rlist0p` recognizes an rlist of which every member is `(r0)`;
- `rlistn0` returns an rlist of a specified length of which every member is `(r0)`;
- `rlist-sum` and `rlist-prod` compute the sum and product, respectively, of the members of an rlist;
- `rlist-scalar-mul` multiplies each member of an rlist by a given scalar and returns a list of the products;
- `rdot` computes the *dot product* of two rlists of the same length, i.e., the sum of the products of corresponding members;
- `rdot-list` returns the list of dot products of an rlist with the members of a list of rlists.

The reader may anticipate that a function name containing the character `r`, suggesting *ring*, is likely to have an analog in Part II with `r` replaced by `f`, suggesting *field*.

3 Matrices

The ACL2 events reported in this section are taken from the file `rmat.lisp`, which begins with the definition of an `m`×`n` *matrix* `a` over the ring `R` as a proper list of `m` rlists, each of length `n`:

```
(defun rmatp (a m n)
  (if (zp m)
      (null a)
    (and (consp a)
         (rlistnp (car a) n)
         (rmatp (cdr a) (1- m) n))))
```

Each member of `a` is a *row*; a *column* is constructed by extracting an entry from each row:

```
(defun row (i a) (nth i a))
(defun col (j a)
  (if (consp a)
      (cons (nth j (car a)) (col j (cdr a)))
    ()))
```

The entry of `a` in row `i` and column `j`:

```
(defun entry (i j a) (nth j (nth i a)))
```

The basic operation of replacing row `k` of `a` with an rlist `r`:

```
(defun replace-row (a k r)
  (if (zp k)
      (cons r (cdr a))
    (cons (car a) (replace-row (cdr a) (1- k) r))))
```

If two m×n matrices are not equal, then some pair of corresponding entries are different. The function entry-diff conducts a search and returns the row and column in which this occurs:

```
(defthmd rmat-entry-diff-lemma
  (implies (and (posp m) (posp n) (rmatp a m n) (rmatp b m n) (not (equal a b)))
           (let* ((pair (entry-diff a b)) (i (car pair)) (j (cdr pair)))
             (and (natp i) (< i m) (natp j) (< j n)
                  (not (equal (entry i j a) (entry i j b)))))))
```

If we can prove that corresponding entries of a and b are equal, then we may invoke this result to conclude that a = b.

The recursive definitions of the sum of two matrices, (rmat-add a b), and the product of a scalar and a matrix, (rmat-scalar-mul c a), are trival. We shall also find it convenient to define the sum of the entries of a matrix in row-major order:

```
(defun rmat-sum (a)
  (if (consp a)
      (r+ (rlist-sum (car a)) (rmat-sum (cdr a)))
    (r0)))
```

Matrix multiplication is a more complicated operation, deferred to Subsection 3.2.

3.1 Transpose

The *transpose* of a matrix is the list of its columns:

```
(defun transpose-mat-aux (a j n)
  (if (and (natp j) (natp n) (< j n))
      (cons (col j a) (transpose-mat-aux a (1+ j) n))
    ()))
(defund transpose-mat (a) (transpose-mat-aux a 0 (len (car a))))
```

We list some simple consequences of the definition:

```
(defthm transpose-rmat-entry
  (implies (and (posp m) (posp n) (rmatp a m n) (natp j) (< j n) (natp i) (< i m))
           (equal (entry j i (transpose-mat a))
                  (entry i j a))))
(defthm transpose-rmat-2
  (implies (and (posp m) (posp n) (rmatp a m n))
           (equal (transpose-mat (transpose-mat a))
                  a)))
(defthmd col-transpose-rmat
  (implies (and (posp m) (posp n) (rmatp a m n) (natp j) (< j m))
           (equal (col j (transpose-mat a))
                  (row j a))))
```

The replacement of a column is now readily defined using the transpose:

```
(defund replace-col (a k r) (transpose-mat (replace-row (transpose-mat a) k r)))
```

Our proof of associativity of matrix multiplication uses the observation that the entries of an m×n matrix a have the same sum, as computed by rmat-sum, as those of its transpose. This is trivially true if either m or n is 0. Otherwise, we derive the (m-1)x(n-1) matrix (strip-mat a) by deleting the first row and the first column of a, and prove the following:

```
(defthmd sum-rmat-strip-mat
  (implies (and (posp m) (posp n) (rmatp a m n))
           (equal (rmat-sum a)
                  (r+ (entry 0 0 a)
                      (r+ (r+ (rlist-sum (cdr (row 0 a)))
                              (rlist-sum (cdr (col 0 a))))
                          (rmat-sum (strip-mat a)))))))
```

The desired lemma follows by induction, using `sum-rmat-strip-mat` to rewrite both sides of the equation and `col-transpose-rmat` to complete the proof:

```
(defthmd sum-rmat-transpose
  (implies (and (natp m) (natp n) (rmatp a m n))
           (equal (rmat-sum (transpose-mat a))
                  (rmat-sum a))))
```

3.2 Multiplication

The product of matrices `a` and `b` is defined when the number of columns of `a` is the number of rows of `b`. The product has the same number of rows as `a` and the same number of columns as `b`. Each row of the product is the list of dot products of the corresponding row of `a` and the columns of `b`:

```
(defund rmat* (a b)
  (if (consp a)
      (cons (rdot-list (car a) (transpose-mat b))
            (rmat* (cdr a) b))
    ()))
(defthm rmatp-rmat*
  (implies (and (rmatp a m n) (rmatp b n p) (posp m) (posp n) (posp p))
           (rmatp (rmat* a b) m p)))
(defthmd rmat*-entry
  (implies (and (posp m) (posp n) (posp p) (rmatp a m n) (rmatp b n p)
                (natp i) (< i m) (natp j) (< j p))
           (equal (entry i j (rmat* a b))
                  (rdot (row i a) (col j b)))))
```

The formula for the transpose of a product is an immediate consequence of `transpose-rmat-entry`, `rmat*-entry`, and `rmat-entry-diff-lemma`:

```
(defthmd transpose-rmat*
  (implies (and (posp m) (posp n) (posp p) (rmatp a m n) (rmatp b n p))
           (equal (transpose-mat (rmat* a b))
                  (rmat* (transpose-mat b) (transpose-mat a)))))
```

For `i` $<$ `n`, row `i` of the `n`$\times$`n` *identity matrix* is the *unit vector* `(runit i n)`, the rlist of length `n` with 1 at index `i` and 0 elsewhere:

```
(defun runit (i n)
  (if (zp n) ()
    (if (zp i) (cons (r1) (rlistn0 (1- n)))
      (cons (r0) (runit (1- i) (1- n))))))
(defun id-rmat-aux (i n)
  (if (and (natp i) (natp n) (< i n))
```

```
      (cons (runit i n) (id-rmat-aux (1+ i) n))
    ()))
(defund id-rmat (n) (id-rmat-aux 0 n))
```

The entries of the identity matrix are given by the *Kronecker delta* function:

```
(defun rdelta (i j) (if (= i j) (r1) (r0)))
(defthmd entry-id-rmat
  (implies (and (natp n) (natp i) (natp j) (< i n) (< j n))
           (equal (entry i j (id-rmat n)) (rdelta i j))))
```

It follows that the identity matrix is its own transpose, which in turn implies its defining properties:

```
(defthmd transpose-id-rmat
  (implies (natp n) (equal (transpose-mat (id-rmat n)) (id-rmat n))))
(defthmd id-rmat-right
  (implies (and (posp m) (posp n) (rmatp a m n))
           (equal (rmat* a (id-rmat n)) a)))
(defthmd id-rmat-left
  (implies (and (posp m) (posp n) (rmatp a m n))
           (equal (rmat* (id-rmat m) a) a)))
```

To prove associativity of multiplication, let `a`, `b`, and `c` be matrices of dimensions `m`×`n`, `n`×`p`, and `p`×`q`, respectively, so that both products `(rmat a (rmat* b c))` and `(rmat* (rmat* a b) c)` are `m`×`q` matrices. It will suffice to show that corresponding entries agree:

```
(entry i j (rmat* a (rmat* b c))) = (entry i j (rmat* (rmat* a b) c)).
```
(1)

The usual informal proof proceeds by expanding the matrix products as well as the resulting dot products. In standard notation (e.g., writing a_{ir} for `(entry i r a)`), the resulting goal is

$$\sum_{r=0}^{n-1}\sum_{s=0}^{p-1} a_{ir}b_{rs}c_{sj} = \sum_{s=0}^{p-1}\sum_{r=0}^{n-1} a_{ir}b_{rs}c_{sj}.$$

The proof is completed by simply observing that the sum on the right is a rearrangement of the three-way products that appear in the sum on the left. Our objective is a formal proof that captures the intuition underlying this observation.

We shall show that these products are the entries of the `n`×`p` matrix `(rmat12 a b c i j)`, defined as follows:

```
(defun rlist-mul-list (x l)
  (if (consp l)
      (cons (rlist-mul x (car l))
            (rlist-mul-list x (cdr l)))
    ()))
(defun rlist-scalar-mul-list (x l)
  (if (consp l)
      (cons (rlist-scalar-mul (car x) (car l))
            (rlist-scalar-mul-list (cdr x) (cdr l)))
    ()))
(defund rmat12 (a b c i j)
  (rlist-scalar-mul-list (row i a) (rlist-mul-list (col j c) b)))
```

To compute the entries of this matrix, first we compute its `r`th row:

```
(nth r (rmat12 a b c i j))
  = (rlist-scalar-mul (nth r (row i a)) (nth r (rlist-mul-list (col j c) b)))
  = (rlist-scalar-mul (entry i r a) (rlist-mul (col j c) (nth r b)))
```

Now the `s`th entry of the `r`th row:

```
(entry r s (rmat12 a b c i j))
  = (nth s (nth r (rmat12 a b c i j)))
  = (nth s (rlist-scalar-mul (entry i r a) (rlist-mul (col j c) (nth r b))))
  = (entry i r a) * (nth s (rlist-mul (col j c) (nth r b)))
  = (entry i r a) * ((nth s (col j c)) * (nth s (nth r b)))
  = (entry i r a) * ((entry s j c) * (entry r s b))
  = (entry i r a) * (entry r s b) * (entry s j c)
```

Next we compute `(rmat-sum (rmat12 a b c i j))`. As a first step, it is easily shown by induction that if `x` is an rlist of length `n` and `l` is a matrix with `n` rows, then

```
(rmat-sum (rlist-scalar-mul-list x l)) = (rdot x (rlist-sum-list l)).
```

We apply this result to the definition of `rmat-sum`, substituting `(row i a)` for `x` and `(rlist-mul-list (col j c) b)` for `l`. This yields the following expression for `rmat-sum (rmat12 a b c i j)`:

```
(rdot (row i a) (rlist-sum-list (rlist-mul-list (col j c) b))).
```

Note that `(rlist-sum-list (rlist-mul-list (col j c) b))` and `(col j (rmat* b c))` are both rlists of length `n`. To prove equality, it suffices to show that corresponding members are equal:

```
(nth k (rlist-sum-list (rlist-mul-list (col j c) b)))
  = (rlist-sum (nth k (rlist-mul-list (col j c) b)))
  = (rlist-sum (rlist-mul (col j c) (nth k b)))
  = (rdot (col j c) (nth k b))
  = (rdot (col j c) (row k b))
  = (rdot (row k b) (col j c))
  = (entry k j (rmat* b c))
  = (nth k (col j (rmat* b c)))
```

Thus, `(rlist-sum-list (rlist-mul-list (col j c) b)) = (col j (rmat* b c))`. It follows that

```
(rmat-sum (rmat12 a b c i j)) = (rdot (row i a) (col j (rmat* b c)))
                              = (entry i j (rmat* a (rmat* b c))):
```

The `p`×`n` matrix corresponding to the right side of Equation (1) is similarly defined:

```
(defund rmat21 (a b c i j)
  (rlist-scalar-mul-list (col j c)
                         (rlist-mul-list (row i a) (transpose-mat b))))
```

Minor variations in the above derivations yield an expression for the entries of this matrix,

```
(entry r s (rmat21 a b c i j)) = (r* (entry i s a) (r* (entry s r b) (entry r j c))),
```

and the sum of these entries:

```
(rmat-sum (rmat21 a b c i j)) = (entry i j (rmat* (rmat* a b) c)).
```

Thus, `(entry r s (rmat21 a b c i j)) = (entry s r (rmat12 a b c i j))`, and hence

```
(transpose-mat (rmat12 a b c i j)) = (rmat21 a b c i j).
```

Finally, Equation (1) follows from `sum-rmat-transpose`, and associativity holds:

```
(defthmd rmat*-assoc
  (implies (and (rmatp a m n) (rmatp b n p) (rmatp c p q)
                (posp m) (posp n) (posp p) (posp q))
           (equal (rmat* a (rmat* b c))
                  (rmat* (rmat* a b) c))))
```

4 Determinants

In `rdet.lisp`, we formalize the classical definition of the *determinant* of an `n`×`n` matrix over the ring `R`, based on the symmetric group `(sym n)` as defined in `books/projects/groups/symmetric.lisp` and documented in [13]. The elements of this group are the members of the list `(slist n)` of permutations of the list `(ninit n)` = `(0 1 ... n-1)`. Such a permutation `p` may be viewed as a bijection of `(ninit n)` that maps an index `j` to `(nth j p)`. The composition of permutations `p` and `q` is computed by the group operation, `(comp-perm p q n)`. Note that `(ninit n)` itself is the group identity.

A *transposition* is a permution, denoted by `(transpose i j n)`, that simply interchanges two distinct indices `i` and `j`. Every permutation may be represented as a composition of a list of transpositions, and while neither this list nor its length is unique, its length is either always even or always odd for a given permutation `p`; `p` is said to be *even* or *odd* accordingly.

A permutation `p` is applied to an arbitrary list `l` of length `n` by the following function:

```
(defun permute (l p)
  (if (consp p)
      (cons (nth (car p) l) (permute l (cdr p)))
    ()))
```

A critical property of `permute` pertains to a product of permutations:

```
(defthm permute-comp-perm
  (implies (and (true-listp l) (consp l) (in x (sym (len l))) (in y (sym (len l))))
           (equal (permute (permute l x) y)
                  (permute l (comp-perm x y (len l))))))
```

Each permutation `p` in `(sym n)` contributes a term `(rdet-term a p n)` to the determinant of an `n`×`n` matrix `a`, computed as follows:

(1) For each `i` < `n`, select the entry of `(row i a)` in column `(nth i p)`;

(2) Compute the product of these `n` entries;

(3) Negate the product if `p` is an odd permutation.

```
(defun rdet-prod (a p n)
  (if (zp n)
      (r1)
    (r* (rdet-prod a p (1- n))
        (entry (1- n) (nth (1- n) p) a))))
(defund rdet-term (a p n)
  (if (even-perm-p p n)
      (rdet-prod a p n)
    (r- (rdet-prod a p n))))
```

The determinant of `a` is the the sum over `(slist n)` of these signed products:

```
(defun rdet-sum (a l n)
  (if (consp l)
      (r+ (rdet-term a (car l) n) (rdet-sum a (cdr l) n))
    (r0)))
(defund rdet (a n) (rdet-sum a (slist n) n))
```

4.1 Properties

To compute the determinant of the identity matrix, note that if p is any permutation other than the identity (ninit n), we can find i < n such that (nth i p) ≠ i, and hence (entry i (nth i p) (id-rmat n)) = 0, which implies (rdet-term (id-rmat n) p n) = 0. On the other hand, (nth i (ninit n)) = i for all i, which implies (rdet-term (id-rmat n) (ninit n) n) = 1. Thus,

```
(defthm rdet-id-rmat (implies (posp n) (equal (rdet (id-rmat n) n) (r1))))
```

The determinant is invariant under transpose-mat. This follows from the observation that the term contributed to the determinant of the transpose of a by a permutation p is the same as the term contributed to the determinant of a by the inverse of p:

```
(defthmd rdet-transpose
  (implies (and (posp n) (rmatp a n n))
           (equal (rdet (transpose-mat a) n) (rdet a n))))
```

If every entry of the kth row of a is 0, then for all p, the kth factor of (rdet-prod a p n) is 0, and it follows that the determinant of a is 0:

```
(defthmd rdet-row-0
  (implies (and (rmatp a n n) (posp n) (natp k) (< k n) (= (nth k a) (rlistn0 n)))
           (equal (rdet a n) (r0))))
```

Furthermore, the determinant is *alternating*, i.e., if two rows of a are equal, then its determinant is 0. To prove this, suppose rows i and j are equal, where i ≠ j. Given a permutation p, let p' = (comp-perm p (transpose i j n) n). The factors of (rdet-prod a p' n) are the same as those of (rdet-prod a p n). But p and p' have opposite parities, and therefore (rdet-term a p' n) is the negative of (rdet-term a p n). Consequently, the sum of terms contributed by the odd permutations is the negative of the sum of terms contributed by the even permutations, and we have

```
(defthmd rdet-alternating
  (implies (and (rmatp a n n) (posp n)
                (natp i) (< i n) (natp j) (< j n) (not (= i j))
                (= (row i a) (row j a)))
           (equal (rdet a n) (r0))))
```

The determinant is also *n-linear*, i.e., linear as a function of each row. This property is specified in terms of the replace-row operation. For a given row i and permutation p, the term contributed by p to the determinant of (replace-row a i x) is a linear function of x:

```
(defthm rdet-term-replace-row
  (implies (and (rmatp a n n) (posp n) (member p (slist n))
                (rlistnp x n) (rlistnp y n) (rp c)
                (natp i) (< i n))
           (let ((a1 (replace-row a i x))
                 (a2 (replace-row a i y))
                 (a3 (replace-row a i (rlist-add (rlist-scalar-mul c x) y))))
             (equal (rdet-term a3 p n)
                    (r+ (r* c (rdet-term a1 p n)) (rdet-term a2 p n))))))
```

The desired result follows by summing over all permutations:

```
(defthm rdet-n-linear
  (implies (and (rmatp a n n) (posp n) (natp i) (< i n)
                (rlistnp x n) (rlistnp y n) (rp c))
           (equal (rdet (replace-row a i (rlist-add (rlist-scalar-mul c x) y)) n)
                  (r+ (r* c (rdet (replace-row a i x) n))
                      (rdet (replace-row a i y) n)))))
```

4.2 Uniqueness

We shall show that rdet is the unique n-linear alternating function on n×n matrices that satisfies (rdet (id-rmat n) n) = 1. To that end, we introduce a constrained function rdet0 as follows:

```
(encapsulate (((rdet0 * *) => *))
  (local (defun rdet0 (a n) (rdet a n)))
  (defthm rp-rdet0
    (implies (and (rmatp a n n) (posp n))
             (rp (rdet0 a n))))
  (defthmd rdet0-n-linear
    (implies (and (rmatp a n n) (posp n) (natp i) (< i n)
                  (rlistnp x n) (rlistnp y n) (rp c))
             (equal (rdet0 (replace-row a i (rlist-add (rlist-scalar-mul c x) y)) n)
                    (r+ (r* c (rdet0 (replace-row a i x) n))
                        (rdet0 (replace-row a i y) n)))))
  (defthmd rdet0-adjacent-equal
    (implies (and (rmatp a n n) (posp n)
                  (natp i) (< i (1- n)) (= (row i a) (row (1+ i) a)))
             (equal (rdet0 a n) (r0)))))
```

Our main objective is to prove that

```
(rdet0 a n) = (r* (rdet a n) (rdet0 (id-rmat n))).
```
(2)

If we then prove that a given function (f a n) satisfies the constraints on rdet0, then we may conclude by functional instantiation that (f a n) = (r* (rdet a n) (f (id-rmat n) n)). From this it will follow that if f has the additional property (f (id-rmat n) n) = 1, then (f a n) = (rdet a n).

Note that instead of assuming that rdet0 is alternating, we have imposed the weaker constraint rdet0-adjacent-equal, which says that the value is 0 if two *adjacent* rows are equal. This relaxes the proof obligations for functional instantiation, which will be critical for the proof of correctness of cofactor expansion (Section 5). However, it is a consequence of the above constraints that rdet0 is alternating. To establish this, we first show by a sequence of applications of rdet0-n-linear and rdet0-adjacent-equal that transposing two adjacent rows negates the value of rdet0. It is also easily shown that an arbitrary transposition may be expressed as a composition of an odd number of transpositions of adjacent rows, and it follows that the value is negated by transposing any two rows:

```
(defthmd rdet0-permute-transpose
  (implies (and (rmatp a n n) (posp n)
                (natp i) (natp j) (< i j) (< j n))
           (equal (rdet0 (permute a (transpose i j n)) n)
                  (r- (rdet0 a n)))))
```

Since every permutation is a product of transpositions, this yields the following generalization:

```
(defthmd rdet0-permute-rows
  (implies (and (rmatp a n n) (posp n) (in p (sym n)))
           (equal (rdet0 (permute a p) n)
                  (if (even-perm-p p n)
                      (rdet0 a n)
                    (r- (rdet0 a n))))))
```

Now suppose (row i a) = (row j a), where $0 \leq$ i $<$ j $<$ n. By rdet0-adjacent-equal, we may also assume i + 1 $<$ j. Let a' = (permute (transpose (1+ i) j n) a). Then

```
(nth (1+ i) a') = (nth j a) = (nth i a) = (nth i a').
```

By `rdet0-adjacent-equal`, `(rdet0 a')` = 0, and by `rdet0-permute-transpose`,

```
(rdet0 a n) = (r- (rdet0 a' n) = (r- 0) = 0.
```

Thus, `rdet0` is an alternating function:

```
(defthmd rdet0-alternating
  (implies (and (rmatp a n n) (posp n) (natp i) (natp j) (< i n) (< j n)
                (not (= i j)) (= (row i a) (row j a)))
           (equal (rdet0 a n) (r0))))
```

Our proof of Equation (2) involves arbitrary lists of length `k` $\leq$ `n` of natural numbers less than `n`, which we call `k`-*tuples*. We begin with the following definitions:

- `(tuplep x k n)` is a predicate that recognizes a `k`-tuple;
- `(extend-tuple x n)` returns the list of `n` (`k`+1)-tuples constructed from a given `k`-tuple `x` by appending each natural number less than `n`;
- `(extend-tuples l n)` returns the list of all (`k`+1)-tuples constructed in this way from the members of a list `l` of `k`-tuples.

The list of all `k`-tuples is defined recursively:

```
(defun all-tuples (k n)
  (if (zp k)
      (list ())
    (extend-tuples (all-tuples (1- k) n) n)))
```

Let `a` be a fixed `n`×`n` matrix. We associate a value `(reval-tuple x k a n)` with each `k`-tuple `x` as follows. First we construct an rlist of length `k`, `(extract-entries x a)`, the `j`th member of which is `(entry j (nth j x) a)`:

```
(defun extract-entries (x a)
  (if (consp x)
      (cons (nth (car x) (car a))
            (extract-entries (cdr x) (cdr a)))
    ()))
```

We define `(runits x n)` to be the list of unit vectors corresponding to the members of `x`:

```
(defun runits (x n)
  (if (consp x)
      (cons (runit (car x) n) (runits (cdr x) n))
    ()))
```

The value `(reval-tuple x k a n)` is the product of the members of `(extract-entries x a)` together with the value of `rdet0` applied to the matrix derived from `a` by replacing its first `k` rows with `(runits x n)`:

```
(defun reval-tuple (x k a n)
  (r* (rlist-prod (extract-entries x a))
      (rdet0 (append (runits x n) (nthcdr k a)) n)))
```

We also define the sum of the values of `(reval-tuple x k a n)` as `x` ranges over a list `l` of `k`-tuples:

```
(defun rsum-tuples (l k a n)
  (if (consp l)
      (r+ (reval-tuple (car l) k a n) (rsum-tuples (cdr l) k a n))
    (r0)))
```

We would like to compute `(rsum-tuples (all-tuples k n) k a n)`. Since the only member of `(all-tuples 0 n)` is `NIL`, the case k = 0 is trivial:

```
(rsum-tuples (all-tuples 0 n) 0 a n) = (reval-tuple () 0 a n) = (rdet0 a n).    (3)
```

For the case k = n, we observe that `(nthcdr n a)` = `NIL` and that if the members of `x` are not distinct, then the matrix `(runits x n)` has two equal rows and by `rdet0-alternating`, `(rdet0 (runits x n) n)` = 0. Thus, in the computation of `(rsum-tuples (all-tuples n n) n a n)`, we need only consider the n-tuples that are permutations. If `p` is in `(sym n)`, then by `rdet0-permute-rows`,

```
(rdet0 (runits p n) n)
  = (rdet0 (permute (id-rmat n) p) n)
  = (if (even-perm-p n n) (rdet0 (id-rmat n) n) (r- (rdet0 (id-rmat n) n)))
```

and `(extract-entries p a)` = `(rdet-prod a p n)`. Consequently,

```
(reval-tuple p n a n) = (r* (rdet-term a p n) (rdet0 (id-rmat n) n)).
```

Summing over `(slist n)`, we have

```
(rsum-tuples (all-tuples n n) n a n) = (r* (rdet a n) (rdet0 (id-rmat n) n)).    (4)
```

For $0 \leq k < n$ and `(tuplep x k n)`, repeated application of `rdet0-n-linear` yields

```
(rsum-tuples (extend-tuple x) (1+ k) a n) = (reval-tuple x k a n).
```

Summing over `(all-tuples k n)`, we have the recurrence formula

```
(rsum-tuples (all-tuples (1+ k) n) (1+ k) a n) = (rsum-tuples (all-tuples k n) k a n).
```

By induction, `(rsum-tuples (all-tuples k n) k a n)` is independent of k. In particular,

```
(rsum-tuples (all-tuples n n) n a n) = (rsum-tuples (all-tuples 0 n) 0 a n).
```

Equation (2) follows from this result together with Equations (3) and (4):

```
(defthmd rdet-unique
  (implies (rmatp a n n)
           (equal (rdet0 a n)
                  (r* (rdet a n) (rdet0 (id-rmat n) n)))))
```

4.3 Multiplicativity

If we had further constrained the function `rdet0` to satisfy `(rdet0 (id-rmat n) n)` = 1, then we could have replaced the conclusion of `rdet-unique` with the simpler equation `(rdet0 a n)` = `(rdet a n)`. One reason behind our weaker specification is that it allows us to prove the multiplicativity property, `(rdet (rmat* a b) n)` = `(r* (rdet a n) (rdet b n)`, by functional instantiation. We define

```
(defun rdet-rmat* (a b n) (rdet (rmat* a b) n))
```

Our goal is the functional instance of `rdet-unique` derived by substituting

```
(lambda (a n) (rdet-rmat* a b n))
```

for `rdet0`. This requires that we prove the analogs of the two nontrivial constraints on `rdet0`. The first is a consequence of `rdet-n-linear` and the definitions of `rmat*`, `rdot-list`, and `rlist-scalar-mul`:

```
(defthmd rdet-rmat*-n-linear
  (implies (and (rmatp a n n) (rmatp b n n) (posp n) (natp k) (< k n)
                (rlistnp x n) (rlistnp y n) (rp c))
           (equal (rdet-rmat* (replace-row a k (rlist-add (rlist-scalar-mul c x) y))
                              b n)
                  (r+ (r* c (rdet-rmat* (replace-row a k x) b n))
                      (rdet-rmat* (replace-row a k y) b n)))))
```

The second follows from `rdet-alternating` and the observation that if `(row k a) = (row (1+ k) a)`, then `(row k (rmat* a b)) = (row (1+ k) (rmat* a b))`:

```
(defthmd rdet-rmat*-adjacent-equal
  (implies (and (rmatp a n n) (rmatp b n n) (posp n)
                (natp k) (< k (1- n)) (= (row k a) (row (1+ k) a)))
           (equal (rdet-rmat* a b n) (r0))))
```

Functional instantiation of `rdet-unique` yields

```
(rdet-rmat* a b n) = (r* (rdet a n) (rdet-rmat* (id-rmat n) b n)).
```

Expanding `rdet-rmat*` and applying `id-rmat-left`, we have

```
(defthmd rdet-multiplicative
  (implies (and (rmatp a n n) (rmatp b n n) (posp n))
           (equal (rdet (rmat* a b) n)
                  (r* (rdet a n) (rdet b n)))))
```

5 Cofactors

Given an `n`×`n` matrix `a`, we define the `(n-1)`×`(n-1)` submatrix `(minor i j a)` to be the result of deleting the `i`th row and the `j`th column of `a`:

```
(defun delete-row (k a)
  (if (zp k) (cdr a)
    (cons (car a) (delete-row (1- k) (cdr a)))))
(defund delete-col (k a) (transpose-mat (delete-row k (transpose-mat a))))
(defund minor (i j a) (delete-col j (delete-row i a)))
```

Its entries may be computed as follows:

```
(defthmd entry-rmat-minor
  (implies (and (rmatp a n n) (natp n) (> n 1) (natp i) (natp j) (< i n) (< j n)
                (natp r) (natp s) (< r (1- n)) (< s (1- n)))
           (equal (entry r s (minor i j a))
                  (entry (if (>= r i) (1+ r) r) (if (>= s j) (1+ s) s) a))))
```

The *cofactor* of an entry of `a` is the determinant of its minor with an attached sign determined by the parity of the sum of its indices:

```
(defund rdet-cofactor (i j a n)
  (if (evenp (+ i j))
      (rdet (minor i j a) (1- n))
    (r- (rdet (minor i j a) (1- n)))))
```

5.1 Cofactor Expansion

The cofactor expansion of the determinant of `a` by a column is computed by multiplying each entry of the column by its cofactor and summing the products:

```
(defun expand-rdet-col-aux (a i j n)
  (if (zp i) (r0)
    (r+ (r* (entry (1- i) j a) (rdet-cofactor (1- i) j a n))
        (expand-rdet-col-aux a (1- i) j n))))
(defund expand-rdet-col (a j n) (expand-rdet-col-aux a n j n))
```

Cofactor expansion by a row is similarly defined:

```
(defun expand-rdet-row-aux (a i j n)
  (if (zp j) (r0)
    (r+ (r* (entry i (1- j) a) (rdet-cofactor i (1- j) a n))
        (expand-rdet-row-aux a i (1- j) n))))
(defund expand-rdet-row (a i n) (expand-rdet-row-aux a i n n))
```

It follows from `entry-rmat-minor` and `transpose-rmat-entry` that

```
(transpose-mat (minor i j a)) = (minor j i (transpose-mat a)),
```

which, in combination with `rdet-transpose`, implies

```
(rdet-cofactor j i (transpose-mat a) n) = (rdet-cofactor i j a n).
```

Consequently, cofactor expansion by column `i` is equivalent to expansion of the transpose by row `i`:

```
(defthmd expand-rdet-row-transpose
  (implies (and (rmatp a n n) (natp n) (> n 1) (natp i) (< i n))
           (equal (expand-rdet-row (transpose-mat a) i n)
                  (expand-rdet-col a i n))))
```

We shall prove, by functional instantiation of `rdet-unique`, that the result of cofactor expansion by a column has the same value as the determinant, and it will follow that the same is true for expansion by a row. Once again, this requires proving analogs of the constraints on `rdet0`.

It is clear that replacing row `i` of `a` does not alter `(rdet-cofactor i j a b)`. On the other hand, for $\mathtt{k} \neq \mathtt{i}$, `(rdet-cofactor i j a n)` is a linear function of `(row k a)`:

```
(defthmd rdet-cofactor-n-linear
  (implies (and (rmatp a n n) (natp n) (> n 1) (natp i) (< i n) (natp j) (< j n)
                (natp k) (< k n) (not (= k i)) (rlistnp x n) (rlistnp y n) (rp c))
           (equal (rdet-cofactor
                     i j (replace-row a k (rlist-add (rlist-scalar-mul c x) y)) n)
                  (r+ (r* c (rdet-cofactor i j (replace-row a k x) n))
                      (rdet-cofactor i j (replace-row a k y) n)))))
```

It follows that cofactor expansion by column `j` is `n`-linear:

```
(defthmd expand-rdet-col-n-linear
  (implies (and (rmatp a n n) (natp n) (> n 1) (natp j) (< j n)
                (natp k) (< k n) (rlistnp x n) (rlistnp y n) (rp c))
           (equal (expand-rdet-col
                    (replace-row a k (rlist-add (rlist-scalar-mul c x) y)) j n)
                  (r+ (r* c (expand-rdet-col (replace-row a k x) j n))
                      (expand-rdet-col (replace-row a k y) j n)))))
```

Now suppose adjacent rows k and k + 1 are equal. Then for any index i other than k or k + 1, `(minor i j a)` has two equal adjacent rows, and therefore `(rdet-cofactor i j a n)` = 0. Meanwhile,

```
(minor k j) = (minor (1+ k) j)
```

and

```
(entry k j a) = (entry (1+ k) j a),
```

but k + j and (k + 1) + j have opposite parities, and hence

```
(rdet-cofactor k j a n) + (rdet-cofactor (1+ k) j a n) = 0.
```

Therefore, `(expand-rdet-col a j n)` = 0:

```
(defthmd expand-rdet-col-adjacent-equal
  (implies (and (rmatp a n n) (> n 1) (natp j) (< j n)
                (natp k) (< k (1- n)) (= (row k a) (row (1+ k) a)))
           (equal (expand-rdet-col a j n) (r0))))
```

Thus, the constraints on `rdet0` are satisfied, and by functional instantiation of `rdet-unique`, we have the following:

```
(defthmd expand-rdet-col-val
  (implies (and (rmatp a n n) (posp n) (> n 1) (natp k) (< k n))
           (equal (expand-rdet-col a k n)
                  (r* (rdet a n) (expand-rdet-col (id-rmat n) k n)))))
```

It remains to show that `(expand-rdet-col (id-rmat n) k n)` = 1. By `row-rmat-minor` (see `rdet.lisp`), we have the following expression for a row of `(minor i j (id-rmat n))`:

```
(defthmd nth-minor-id-rmat
  (implies (and (natp n) (> n 1) (natp i) (< i n) (natp j) (< j n)
                (natp r) (< r (1- n)))
           (equal (nth r (minor i j (id-rmat n)))
                  (delete-nth j (runit (if (< r i) r (1+ r)) n)))))
```

The following is a consequence of the definitions of `runit` and `delete-nth`:

```
(defthmd delete-nth-runit
  (implies (and (posp n) (natp j) (< j n) (natp k) (< k n))
           (equal (delete-nth j (runit k n))
                  (if (< j k) (runit (1- k) (1- n))
                    (if (> j k) (runit k (1- n))
                      (rlistn0 (1- n)))))))
```

Consequently, if i ≠ j, then we find a zero row of `(minor i j (id-rmat n))`, and by `rdet-row-0`, its determinant is 0. On the other hand, `(minor j j (id-rmat n))` = `(id-rmat (1- n))` and the corresponding cofactor is 1, as is the cofactor expansion:

```
(defthmd expand-rdet-col-id-rmat
  (implies (and (rmatp a n n) (natp n) (> n 1) (natp j) (< j n))
           (equal (expand-rdet-col (id-rmat n) j n) (r1))))
```

Combining this with `expand-rdet-col-val`, we have the correctness theorem for column expansion:

```
(defthmd expand-rdet-col-rdet
  (implies (and (rmatp a n n) (posp n) (> n 1) (natp k) (< k n))
           (equal (expand-rdet-col a k n) (rdet a n))))
```

It follows from `rdet-transpose`, `expand-rdet-row-transpose`, and `transpose-rmat-2` that the same holds for row expansion:

```
(defthmd expand-rdet-row-rdet
  (implies (and (rmatp a n n) (posp n) (> n 1) (natp k) (< k n))
           (equal (expand-rdet-row a k n) (rdet a n))))
```

As a consequence of `expand-rdet-row-rdet`, we have a recursive version of `rdet`, based on cofactor expansion with respect to row 0:

```
(mutual-recursion
  (defund rdet-rec-cofactor (j a n)
    (if (zp n) ()
      (if (evenp j) (rdet-rec (minor 0 j a) (1- n))
        (r- (rdet-rec (minor 0 j a) (1- n))))))
  (defun expand-rdet-rec-aux (a j n)
    (if (zp j) (r0)
      (r+ (r* (entry 0 (1- j) a) (rdet-rec-cofactor (1- j) a n))
          (expand-rdet-rec-aux a (1- j) n))))
  (defund expand-rdet-rec (a n) (expand-rdet-rec-aux a n n))
  (defun rdet-rec (a n)
    (if (zp n) (r0)
      (if (= n 1) (entry 0 0 a)
        (expand-rdet-rec a n)))))
```

The equivalence follows from `expand-rdet-row-rdet` by induction (see `rdet.lisp` for details):

```
(defthmd rdet-rec-rdet
  (implies (and (rmatp a n n) (posp n))
           (equal (rdet-rec a n) (rdet a n))))
```

5.2 Classical Adjoint

We shall define the *cofactor matrix* of an `n`×`n` matrix `a` to be the `n`×`n` matrix with entries

```
(entry i j (cofactor-rmat a b)) = (rdet-cofactor i j a n).
```

To define this matrix, we first define a function that computes its `i`th row:

```
(defun cofactor-rmat-row-aux (i j a n)
  (if (and (natp n) (> n 1) (natp j) (< j n))
      (cons (rdet-cofactor i j a n) (cofactor-rmat-row-aux i (1+ j) a n))
    ()))
(defund cofactor-rmat-row (i a n) (cofactor-rmat-row-aux i 0 a n))

(defun cofactor-rmat-aux (i a n)
  (if (and (natp n) (natp i) (< i n))
      (cons (cofactor-rmat-row i a n) (cofactor-rmat-aux (1+ i) a n))
    ()))
(defund cofactor-rmat (a n) (cofactor-rmat-aux 0 a n))
```

The *classical adjoint* of a is the transpose of its cofactor matrix:

```
(defund adjoint-rmat (a n) (transpose-mat (cofactor-rmat a n)))
```

The following is an equivalent formulation:

```
(defthmd cofactor-rmat-transpose
  (implies (and (rmatp a n n) (natp n) (> n 1))
           (equal (cofactor-rmat (transpose-mat a) n)
                  (adjoint-rmat a n))))
```

Note that the dot product of `(row i a)` with `(cofactor-rmat-row i a n)` is a rearrangement of the sum `(expand-rdet-row a i n)`:

```
(defthmd rdot-cofactor-rmat-row-expand-rdet-row
  (implies (and (rmatp a n n) (natp n) (> n 1) (natp i) (< i n))
           (equal (rdot (row i a) (cofactor-rmat-row i a n))
                  (expand-rdet-row a i n))))
```

Combining this with `expand-rdet-row-rdet`, we have the following expression for the determinant:

```
(defthmd rdot-cofactor-rmat-row-rdet
  (implies (and (rmatp a n n) (natp n) (> n 1) (natp i) (< i n))
           (equal (rdot (row i a) (cofactor-rmat-row i a n))
                  (rdet a n))))
```

Next we consider the result of substituting `(replace-row a i (row k a))` for a in `rdot-cofactor-rmat-row-rdet`, where $\mathtt{k} \neq \mathtt{i}$. Since this matrix has two identical rows, its determinant is 0, and we have

```
(defthmd rdot-cofactor-rmat-row-rdet-0
  (implies (and (rmatp a n n) (natp n) (> n 1) (natp i) (< i n)
                (natp k) (< k n) (not (= k i)))
           (equal (rdot (row k a) (cofactor-rmat-row i a n))
                  (r0))))
```

Thus, we have the following for general k:

```
(defthmd rdot-cofactor-rmat-row-rdelta
  (implies (and (rmatp a n n) (natp n) (> n 1) (natp i) (< i n) (natp k) (< k n))
           (equal (rdot (row k a) (cofactor-rmat-row i a n))
                  (r* (rdelta i k) (rdet a n)))))
```

Since `(cofactor-rmat-row i a n)` = `(col i (adjoint-mat a n))`, this yields an expression for the n×n matrix product of a and its adjoint:

```
(defthmd rmat*-adjoint-rmat
  (implies (and (rmatp a n n) (natp n) (> n 1))
           (equal (rmat* a (adjoint-rmat a n))
                  (rmat-scalar-mul (rdet a n) (id-rmat n)))))
```

In Part II, where we consider matrices with entries ranging over a field, we shall use this last equation in deriving a criterion for the existence of a multiplicative inverse of a matrix. We shall also apply the results of this subsection to a proof of Cramer's Rule for solving a system of n linear equations in n unknowns.

References

[1] William Brown (1993): *Matrices over Commutative Rings*. M. Dekker.

[2] Ruben Gamboa, John Cowles & Jeff Van Baalen (2003): *Using ACL2 Arrays to Formalize Matrix Algebra*. In: *ACL2 2003: 4th International Workshop on the ACL2 Theorem Prover and its Applications*, Boulder, Colorado.

[3] Joe Hendrix (2003): *Matrices in ACL2*. In: *ACL2 2003: 4th International Workshop on the ACL2 Theorem Prover and its Applications*, Boulder, Colorado.

[4] Kenneth Hoffman & Ray Kunze (1961): *Linear Algebra*. Allyn Prentice-Hall.

[5] Bernard Kolman (1977): *Elementary Linear Algebra*, 2nd edition. MacMillan.

[6] Jin Ho Kwak & Sungpyo Kong (1997): *Linear Algebra*. Birkhäuser.

[7] Carl Kwan & Warren Hunt (2024): *Automatic Verification of Right-greedy Numerical Linear Algebra Algorithms*. In: *Proceedings of the 24th Conference on Formal Methods in Computer-Aided Design (FMCAD 2024)*, doi:10.34727/2024/isbn.978-3-85448-065-5.

[8] Carl Kwan & Warren Hunt (2024): *Formalizing the Cholesky Factorization Theorem*. In: *Proceedings for the Fifteenth Conference on Interactive Theorem Proving (ITP 2024)*, doi:10.4230/LIPIcs.ITP.2024.25.

[9] *Maths in Lean: Linear Algebra*. Available at `https://leanprover-community.github.io/theories/linear_algebra.html`.

[10] Steven Roman (2005): *Advanced Linear Algebra*, 2nd edition. Springer, doi:10.1007/978-1-4757-2178-2.

[11] David M. Russinoff (2022): *A Formalization of Finite Froup Theory*. In: *ACL2 2022: 17th International Workshop on the ACL2 Theorem Prover and its Applications*, Austin, Texas, doi:10.4204/EPTCS.359.10.

[12] David M. Russinoff (2023): *A Formalization of Finite Froup Theory: Part II*. In: *ACL2 2023: 18th International Workshop on the ACL2 Theorem Prover and its Applications*, Austin, Texas, doi:10.4204/EPTCS.393.4.

[13] David M. Russinoff (2023): *A Formalization of Finite Froup Theory: Part III*. In: *ACL2 2023: 18th International Workshop on the ACL2 Theorem Prover and its Applications*, Austin, Texas, doi:10.4204/EPTCS.393.5.

[14] David M. Russinoff (2025): *A Formalization of Elementary Linear Algebra: Part II*. In: *ACL2 2025: 19th International Workshop on the ACL2 Theorem Prover and its Applications*, Austin, Texas.

[15] ZhengPu Shi & Gang Chen (2022): *Integration of Multiple Formal Matrix Models in Coq*. In Wei Dong & Jean-Pierre Talpin, editors: *Dependable Software Engineering Theories, Tools, and Applications*, Springer Nature Switzerland, doi:10.1007/978-3-031-21213-0_11.

[16] ZhengPu Shi & Gang Chen (2024): *Formal Verification of Executable Matrix Inversion via Adjoint Matrix and Gaussian Elimination*. In: *Proceedings of the 26th International Symposium on Principles and Practice of Declarative Programming*, doi:10.1145/3678232.3678242.

[17] Zhiping Shi, Yan Zhang, Zhenke Liu, Ximan Kank, Yong Guan, Jie Zhang & Xiaoyu Song (2014): *Formalization of matrix theory in Hol4*. In: *Advances in Mechanical Engineering*, 6, doi:1155/2014/195276.

[18] Christian Sternagel & Rene Thiemann (2010): *Executable Matrix Operations on Matrices of Arbitrary Dimensions*. In: *Archive of Formal Proofs*.