Using ACL2 Arrays to Formalize Matrix Algebra

by

John Cowles

Ruben Gamboa

Jeff Van Baalen

University of Wyoming

{cowles,ruben,jvb}@cs.uwyo.edu

Supported by NASA grant NAG 2-1570.

Matrix Algebra

Let p and q be positive integers.

A $p \times q$ matrix is a rectangular array of *numbers*,

with p rows and q columns,

$$\begin{pmatrix} m_{1\,1} & \cdots & m_{1\,q} \\ \vdots & \ddots & \vdots \\ m_{p\,1} & \cdots & m_{p\,q} \end{pmatrix}$$

Matrix Operations

- The **sum** of two $p \times q$ matrices is a $p \times q$ matrix.

- The **product** of a $p \times q$ matrix and a $q \times r$ matrix is a $p \times r$ matrix.

- The **scalar** product of a *number* and a $p \times q$ matrix is a $p \times q$ matrix.

- The **transpose** of a $p \times q$ matrix is a $q \times p$ matrix.

- Matrix addition and multiplication are **associative**.

$$(M_1 + M_2) + M_3 = M_1 + (M_2 + M_3)$$

- Matrix addition is **commutative**. Matrix multiplication need not be commutative.

$$M_1 + M_2 = M_2 + M_1$$

- Matrix and scalar multiplication **distribute** over matrix addition.

$$M_1 \cdot (M_2 + M_3) = M_1 \cdot M_2 + M_1 \cdot M_3$$

- There is an unique $p \times q$ **zero** matrix $\mathbf{0}$ such that

$$M + \mathbf{0} = M = \mathbf{0} + M.$$

- For *square* matrices, there is an unique $p \times p$ **identity** matrix $\mathbf{I}$ such that

$$M \cdot \mathbf{I} = M = \mathbf{I} \cdot M.$$

- Every $p \times q$ matrix has an unique $p \times q$ **negative** matrix such that

$$M + (-M) = \mathbf{0} = (-M) + M.$$

- Some square matrices, called **nonsingular**, have unique (multiplicative) inverses such that

$$M \cdot M^{-1} = \mathbf{I} = M^{-1} \cdot M.$$

- If $M_1 \cdot M_2 = \mathbf{0}$, then neither M_1 nor M_2 need be **0**.

ACL2 Arrays

ACL2 provides functions for accessing and updating both one and two dimensional arrays,

- with applicative semantics,
- but good access time to the most recently updated copy,
- and usually constant update time.

Should be natural and straight forward to implement the matrix operations using ACL2 two dimensional arrays.

Applicative Semantics for Arrays

- Use a “sparse” array representation.

- An array is an **alist**, i.e. a list of pairs.

  - ◇ One element is the “header” that contains

    - $*$ the number of rows, d_1

    - $*$ the number of columns, d_2

    - $*$ a **default** value

  - ◇ Other elements are of the form $((i\ .\ j)\ .\ \mathbf{val})$.

    - $*$ i and j are integers

    - $*$ $0 \leq i < d_1$ and $0 \leq j < d_2$

    - $*$ **val** is an arbitrary object

Applicative Array Access

To access the value indexed by the pair (i . j) in an array alist:

- Use the function **aref2**
- Search for the first pair whose car matches the pair (i . j).
- If such a pair is found,
  - ◇ then **aref2** returns the cdr of the pair
  - ◇ otherwise **aref2** returns the default value stored in the header.

Fast Array Access

Made possible by maintaining, behind the scenes, a "real" Common Lisp array.

- The real array **may** currently represent the given array alist.
- In that case, an array access can be very fast because the real array can be accessed directly.
- If the real array does **not** currently represent the given array alist, access is done by linear search through the alist:

```
**********************************************
Slow Array Access!  A call of AREF2 on
an array named A1 is being executed
slowly.  See :DOC slow-array-warning
**********************************************
```

Complication

Useful to distinguish two versions of "two dimensional arrays."

Logical or "slow" array.

The **alist** representation used by the applicative semantics.

"Fast" executable array.

A logical array with fast accessing and updating.

Represented, behind the scenes, by a "real" Common Lisp array.

Additional Restriction on "Fast" Arrays

So some compilers can lay down faster code.

Let $d_1 =$ number of rows and
$d_2 =$ number of columns.

Then $d_1 \cdot d_2$ is required to fit into 32 bits.

$$\begin{aligned} d_1 \cdot d_2 &< \textbf{maximum-positive-32-bit-integer} \\ &= 2^{31} - 1 \\ &= 2,147,483,647 \end{aligned}$$

Closure Properties of Matrix Operations

Whenever the results of these operations are mathematically defined,

both **logical** and **fast** arrays are closed under the operations of

- **transpose**,
- **unary-minus**,
- **scalar multiplication**,
- **matrix sum**, and
- **matrix multiplicative inverse**.

Complication due to the

Additional Restriction on "Fast" Arrays

Logical arrays are closed under **matrix product**.

Fast arrays are **not** closed under **matrix product**.

Examples

Suppose the Additional Restriction on "Fast" Arrays is $d_1 \cdot d_2 \leq 20$.

- Product of **fast** arrays need not be **fast**.

$$\begin{array}{ccc} M_1 & \bullet & M_2 \\ 5 \times 2 & & 2 \times 5 \\ & 5 \times 5 & \\ & \uparrow & \end{array}$$

- Two equivalent ways to compute the same **fast** array, with differing results.

$$\begin{array}{ccccc} (M_0 & \bullet & M_1) & \bullet & M_2 \\ 2 \times 5 & & 5 \times 2 & & 2 \times 5 \\ & 2 \times 2 & & & 2 \times 5 \\ & & & 2 \times 5 & \end{array}$$

$$\begin{array}{ccccc} M_0 & \bullet & (M_1 & \bullet & M_2) \\ 2 \times 5 & & 5 \times 2 & & 2 \times 5 \\ 2 \times 5 & & & 5 \times 5 & \\ & & & \uparrow & \end{array}$$

One More Restriction

Ensure that matrix products of **fast** arrays are always **fast** arrays:

Let $d_1 =$ number of rows and
$d_2 =$ number of columns.

$$\begin{aligned} & d_1, d_2 \\ & \quad \leq \lfloor \sqrt{\textbf{maximum-positive-32-bit-integer}} \rfloor \\ & \quad = \lfloor \sqrt{2,147,483,647} \rfloor \\ & \quad = 46,340 \end{aligned}$$

Then $d_1 \cdot d_2$ is guaranteed to be less than the **maximum-positive-32-bit-integer**.

46,340 is **not** enough

`http://www.mat.bham.ac.uk/atlas/v2.0/`

ATLAS of Finite Group Representations

ATLAS: Monster group M

Order =
$2^{46}{\cdot}3^{20}{\cdot}5^{9}{\cdot}7^{6}{\cdot}11^{2}{\cdot}13^{3}{\cdot}17{\cdot}19{\cdot}23{\cdot}29{\cdot}31{\cdot}41{\cdot}47{\cdot}59{\cdot}71$
$\approx 8 \cdot 10^{53}$

Standard generators

Standard generators of the Monster group M are a and b where . . . and ab has order 29.

Update 15th December 1998:
standard generators have now been made as $\mathbf{196{,}882 \times 196{,}882}$ **matrices** over GF(2).
They have been multiplied together, using most of the computing resources of Lehrstuhl D für Mathematik, RWTH Aachen, for about 45 hours.

Test for Matrix Equality: `(m-= M1 M2)`

- **Equivalence** relation on the entire ACL2 universe.

- When one of `M1` or `M2` is **not** a **logical array**, then `m-=` coincides with `equal`.

- **Congruence** relation with respect to the matrix operations of transpose, unary minus, scalar product, sum, and product.

Matrix Sum: `(m-+ M1 M2)`
Matrix Product: `(m-* M1 M2)`

- Defined on the entire ACL2 universe.

- When one of `M1` or `M2` is **not** a logical array, then

  ◇ `m-+` coincides with `+`

  ◇ `m-*` coincides with `*`.

- Allows some hypotheses-free matrix equalities.
  Example **Distributivity**:

```
(m-= (m-* M1 (m-+ M2 M3))
     (m-+ (m-* M1 M2)
          (m-* M1 M3)))
```

- Allows some matrix equalities to use `equal` in place of `m-=`.

- Example: `m-+` satisfies this version of **commutativity**

```
(equal (m-+ M1 M2)
       (m-+ M2 M1))
```

- as well as this weaker version

```
(m-= (m-+ M1 M2)
     (m-+ M2 M1)).
```

- Similar comments apply to the **associativity** of `m-+` and `m-*`.

Determinant
Matrix Inverse: `(m-/ M)`

- Computed using row and column operations.

- Temporary definition:
  A matrix is **nonsingular** iff it is a square matrix and `m-/` does, in fact, compute a two-sided multiplicative inverse.

- Future plans: Use **determinants** to determine if a square matrix is singular or nonsingular.

ACL2 proofs are still required for the following

- For square matrices, whenever the determinant is not 0, then `m-/` computes the two-sided inverse.

- Whenever the determinant is 0 then there is no inverse.

- Non-square matrices do not have two-sided inverses.

```
(defun m-= (M1 M2)
  (declare (xargs :guard
                 (and (array2p '$arg1 M1)
                      (array2p '$arg2 M2))))

  (if (mbt (and (alist2p '$arg1 M1)
                (alist2p '$arg2 M2)))

      (let ((dim1 (dimensions '$arg1 M1))
            (dim2 (dimensions '$arg2 M2)))
           (if (and (= (first dim1)
                       (first dim2))
                    (= (second dim1)
                       (second dim2)))
               (m-=-row-1 (compress2 '$arg1 M1)
                          (compress2 '$arg2
                                     M2)
                          (- (first dim1) 1)
                          (- (second dim1)
                             1))
              nil))

      (equal M1 M2)))
```

- `(array2p name A)` returns `t` if `A` is a two dimensional **fast executable** ACL2 array. Otherwise return `nil`.

  The extra input argument `name` is used by ACL2's "fast" implementation of arrays.

- `(alist2p name A)` returns `t` if `A` is a two dimensional **logical array**. Otherwise return `nil`.

- `(compress2 name A)` allocates and stores **fast array** `A` in a Common Lisp array.

- `(dimensions name A)` returns the dimensions list of the array alist. That list contains d_1 and d_2.

`mbt` ("must be true")

- A new ACL2 Version 2.8 macro.
- Used to replace an expensive Boolean test with `t` during execution.
- Semantically, `(mbt x)` equals `x`
- In raw Lisp `(mbt x)` macro-expands to `t`.
- A guard proof obligation is generated:

  ```
  (implies <guard>
           (equal x t))
  ```

- ACL2's guard verification mechanism ensures that the raw Lisp code is only evaluated when appropriate.

`mbt` ("must be true")

In the definition of `m-=`, since

```
(implies (array2p name M)
         (alist2p name M),
```

the `mbt` replaces the `alist2p` tests with `t` during execution.