# Query languages with structural and analytic properties

Floris Geerts, Thomas Muñoz, Cristian Riveros, Domagoj Vrgoč

## 1 MATLANG syntax and semantics

We assume that we have a supply of matrix variables. The definiton of an instance $I$ on MATLANG is a function defined on a nonempty set $var(I) = \{A, B, M, C, \ldots\}$, that assigns a concrete matrix to each element (matrix *name*) of $var(I)$.

Every expression $e$ is a matrix, either a matrix of $var(I)$ (*base* matrix, if you will) or a result of an operation over matrices.

The syntax of MATLANG expressions is defined by the following grammar. Every sentence is an expression itself.

$$\begin{aligned} e = M & \quad \text{(matrix variable)} \\ \text{let } M = e_1 \text{ in } e_2 & \quad \text{(local binding)} \\ e^* & \quad \text{(conjugate transpose)} \\ \mathbf{1}(e) & \quad \text{(one-vector)} \\ \text{diag}(e) & \quad \text{(diagonalization of a vector)} \\ e_1 \cdot e_2 & \quad \text{(matrix multiplication)} \\ \text{apply}\,[f]\,(e_1, \ldots, e_n) & \quad \text{(pointwise application of } f\text{)} \end{aligned}$$

The operations used in the semantics of the language are defined over complex numbers.

- **Transpose:** if $A$ is a matrix then $A^*$ is its conjugate transpose.
- **One-vector:** if $A$ is a $n \times m$ matrix then $\mathbf{1}(A)$ is the $n \times 1$ column vector full of ones.
- **Diag:** if $v$ is a $m \times 1$ column vector then $\text{diag}(v)$ is the matrix

$$\begin{bmatrix} v_1 & 0 & 0 & \ldots & 0 \\ 0 & v_2 & 0 & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & 0 & \ldots & v_m \end{bmatrix}$$

- **Matrix multiplication:** if $A$ is a $n \times m$ matrix and $B$ is a $m \times p$ matrix then $A \cdot B$ is the $n \times p$ matrix with $(AB)_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$.
- **Pointwise application:** if $A^{(1)}, \ldots, A^{(n)}$ are $m \times p$ matrices, then $\text{apply}[f]\,(A^{(1)}, \ldots, A^{(n)})$ is the $m \times p$ matrix $C$ where $C_{ij} = f(A^{(1)}_{ij}, \ldots, A^{(n)}_{ij})$.

The formal semantics have a set of rules for an expression $e$ to be valid on an instance $I$, this is, $e$ succesfully evaluates to a matrix $A$ on the instance $I$. This success is denoted as $e(I) = A$. Here $I\,[M := A]$ denotes the instance that is equal to $I$ except that maps $M$ to the matrix $A$.

| Expression | Condition for validity |
|---|---|
| $(\text{let } M = e_1 \text{ in } e_2)(I) = B$ | $e_1(I) = A,\ e_2(I\,[M := A]) = B$ |
| $e^*(I) = A^*$ | $e(I) = A$ |
| $\mathbf{1}(e)(I) = \mathbf{1}(A)$ | $e(I) = A$ |
| $\text{diag}(e)(I) = \text{diag}(A)$ | $e(I) = A$, $A$ is a column vector |
| $e_1 \cdot e_2(I) = A \cdot B$ | # columns of $A$ = # rows of $B$ |
| $\text{apply}[f]\,(e_1, \ldots, e_n)(I) = \text{apply}\,[f]\,(A_1, \ldots, A_n)$ | $\forall k, e_k(I) = A$ and all $A_k$ have the same dimentions |

For example,

$$\text{let } N = \mathbf{1}(M)^* \text{ in apply}[c](\mathbf{1}(N)),$$

is an expression, where $c$ denotes the constant function $c : x \to c$. The result is a $1 \times 1$ matrix with $c$ as its entry. This expression is equivalent to $\text{apply}[c](\mathbf{1}(\mathbf{1}(M)^*))$.

An example of what can be computed in MATLANG is the mean of a vector:

$$\begin{aligned}&\text{let } N = \mathbf{1}(v)^* \cdot \mathbf{1}(v) \text{ in}\\ &\text{let } S = v^* \cdot \mathbf{1}(v) \text{ in}\\ &\text{let } R = \text{apply}[\div](S, N) \text{ in } R.\end{aligned}$$

Here, $N$ is the $1 \times 1$ matrix with the dimention of $v$ as its entry. S computes the sum of all the entries of $v$. Finally, in $R$ we store the result of the sum divided by the dimention.

One thing that is worth keeping in mind is that pointwise function application is powerful. With the expression $\text{apply}[f](\cdot)$ one can compute other elemental operations over matrices that can be studied separately, such as:

- Scalar multiplication: we compute $c \cdot A$ as

$$\begin{aligned}&\text{let } C = \text{apply}[c](\mathbf{1}(\mathbf{1}(A)^*)) \text{ in}\\ &\text{let } M = \mathbf{1}(A) \cdot C \cdot \mathbf{1}(A^*)^* \text{ in apply}[\times](M, A).\end{aligned}$$

- Addition: we compute $A + B$ as

$$\text{let apply}[+](A, B).$$

- Trace: let

$$m(x, y) = \begin{cases} x \text{ if } x - y > 0 \\ 0 \text{ if } x - y \leq 0 \end{cases}$$

Then we compute the trace of $A$, $tr(A)$, as

$$\begin{aligned}&\text{let } I = \text{diag}(\mathbf{1}(A)) \text{ in}\\ &\text{let } B = \text{apply}[-](A, I) \text{ in}\\ &\text{let } C = \text{apply}[m](A, B) \text{ in}\\ &\text{let } T = \text{apply}[+](A, I) \text{ in } \mathbf{1}(A)^* \cdot T \cdot \mathbf{1}(A)\end{aligned}$$

## 2 Adding canonical vectors to MATLANG

One thing that we cannot do in MATLANG is to obtain a specific entry of a matrix. This entry is expected to be a $1 \times 1$ matrix. We can do this by adding the standard unit vectors $e_j$ where

$$e_i = \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \rightarrow i\text{-th position}$$

We know show some examples of what can we express with this new feature. For ilustrative reasons, we asume that all the dimentions are well suited for the corresponding operation.

- Get $A_{ij}$ with $e_i^* \cdot A \cdot e_j$.
- The expression $e_i \cdot e_j^*$ is the matrix that has a 1 in the position $i, j$ and zero everywhere else.
- Given a vector $v$, the expression $v \cdot e_i^*$ is the matrix

$$\begin{bmatrix} 0 & \cdots & v_1 & \cdots & 0 \\ 0 & \cdots & v_2 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & v_n & \cdots & 0 \end{bmatrix}$$

- Replace column $j$ of $A$ with zeros: $A(I - e_j \cdot e_j^*)$.
- Replace column $j$ of $A$ with a vector $v$: $A(I - e_j \cdot e_j^*) + v \cdot e_j^*$.

Note that $I = \text{diag}(\mathbf{1}(A))$ and the sum of matrices can be implemented as apply $[+]\,(A, B)$.

### 2.1 Ordering

Let's turn our attention to the order of the canonical vectors $\{v_i\}_i$. If we have the canonical vectors, using a simple matrix product we can compute a function that discriminates if two canonical vectors are the same or not, in particular

$$f(v_i, v_j) = 1 - v_i^* v_j = \begin{cases} 0 \text{ if } i = j \\ 1 \text{ if } i \neq j \end{cases}$$

does exactly that. So, without adding any object, we have equality for canonical vectors. Note that implicitly we are using de identity, the matrix equivalent to the equality relation.

Now, consider the following matrix

$$Z = \begin{bmatrix} 0 & 1 & \cdots & 1 \\ 0 & \ddots & \ddots & \vdots \\ \cdots & \cdots & \cdots & 1 \\ 0 & \cdots & \cdots & 0 \end{bmatrix}.$$

In some way, this matrix gives us an ordering of the canonical vectors. We can see this in the functions of the form

$$f(v_i, v_j) = v_i^* Z v_j = \begin{cases} 1 \text{ if } i < j \\ 0 \text{ if } i \geq j \end{cases}$$

This kind of functions discriminate between canonical vectors, in terms of order. Note that this gives us an answer for any pair of canonical vector, so we have a total order in some sense.

Can we introduce a *local* order of the canonical vectors? For instance, we can ask ourselves if we can define the *successor* or *predecessor* relation of the canonical vectors from a matrix. Let us introduce a new matrix:

$$S := \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & \ddots & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ \cdots & \cdots & \cdots & \ddots & 1 \\ 0 & \cdots & \cdots & \cdots & 0 \end{bmatrix}.$$

Let's see what this matrix does. Note that

$$Sv_i = \begin{cases} v_{i-1} \text{ if } i > 1 \\ \mathbf{0} \text{ if } i = 1 \end{cases}$$

So it's the matrix equivalent of the *predecessor* relation. For example, now we can compute

$$g(v_i, v_j) = v_i^* S v_j = \begin{cases} 1 \text{ if } i = j - 1 \\ 0 \text{ if not} \end{cases}$$

With this matrix we can compute a function that tells us if the argument is the first canonical vector. Note that we don't have exactly

$$\mathbf{1} \cdot Su = 0 \leftrightarrow u = cv_1, \;\; c \in \mathbb{R},$$

it depends on the entries of $u$, but

$$\mathbf{1} \cdot f_{\neq 0}(Su) = 0 \leftrightarrow u = cv_1, \;\; c \in \mathbb{R},$$

and we have that

$$min(u) = (1 - \mathbf{1} \cdot f_{\neq 0}(Su)) = \begin{cases} 1 \text{ if } u = cv_1 \\ 0 \text{ if not} \end{cases}$$

Since $S$ and $Z$ give us an order, it is natural to compute one from the other, so

$$Z = f_{>0}\left(\prod v.(S + I)\right)$$
$$S = Z - f_{>0}(Z^2)$$

# 3 Constant and variable expressions

All the expressions so far are valid if they are composed of matrices that are on the instance $I$ (see section 1). Once you have an expression $E$ that is well defined on the given instance, it doesn't change its value unless the current instance is modified. We call this a *constant* expression.

Now, let's turn our attention on the expressions

$$E(v) = v^* A v.$$

The value of $E$ depends entirely on $v$, we call this a *variable expression*. This type of expressions have *free* variables ($v$ in this case) and it doesn't mean anything in MATLANG unless the *free* variables are mapped into an actual matrix of the current instance, we denote this $E(v)(I[v \to B])$, assuming that $B$ is on the domain of the given instance. This mapping will be accomplished trough operators that are introduced in the next section. These operators define how the variable expression $E$ is used and how it's instantiated. A formal example, let $\{v_i\}_i$ be the canonical vectors. An operator could receive a variable expression $E(v)$ and give the output $E(v_k) = B$, if and only if $[E(v)](I[v \to v_k]) = B$. We now proceed to define the semantics of this formulas.

## 3.1 Extended MATLANG

A schema $S$ is a set of matrix names $M_1, \ldots, M_p$ and a set of vector names $v_1, \ldots, v_l$. An instance $I$ is a function that maps matrix names to concrete matrices (of dimention $n \times n$) and vector names to concrete colun vectors (of dimention $n \times 1$). In addition, we define $dim(I) = n$ and $can(I)$ as the set of canonical vectors of dimention $n$, indexed as $can(I)[1], \ldots, can(I)[n]$.

Let $I$ be an intance and $n = dim(I)$. Let $X$ be a set of vector variables (matrix variables, respectively). A valuation $\nu$ on instance $I$ is a function from $X$ to column vectors of dimention $n \times 1$ (matrices of dimention $n \times n$, respectively).

Let $S$ be an schema, $I$ an instance, $F$ a set of functions $f : \mathbb{R}^k \to \mathbb{R}$ (with $1 \leq k$) and $V$ a set of vector variables. An *extended* MATLANG expression $E$ is given by

$$\begin{aligned} E :=& x \in V | M \in S | v \in S | \\ & | E^* | E_1 + E_2 | E_1 \cdot E_2 | \\ & | f(E_1, \ldots, E_k), f \in F | \\ & | \sum x.E | \prod x.E \end{aligned}$$

Let $n = dim(I)$. We define $\text{type}(E) = (i, j)$ where $(i, j) \in \{1, n\}$. The well typeness is defined recursively as follows.

$$
\begin{aligned}
\text{type}(x) &= (n,1)\\
\text{type}(M) &= (n,n)\\
\text{type}(v) &= (n,1)\\
\text{type}(E^*) &= (j,i) \text{ where } (i,j) = \text{type}(E)\\
\text{type}(E_1 + E_2) &= \begin{cases} \text{type}(E_1) \text{ if } \text{type}(E_1) = \text{type}(E_1) \\ \text{undefined otherwise} \end{cases}\\
\text{type}(E_1 \cdot E_2) &= \begin{cases} (i,k) \text{ if } \text{type}(E_1) = (i,j) \text{ and } \text{type}(E_2) = (j,k) \\ \text{undefined otherwise} \end{cases}\\
\text{type}(f(E_1,\ldots,E_k)) &= \begin{cases} (1,1) \text{ if } \text{type}(E_1) = \cdots = \text{type}(E_k) = (1,1) \\ \text{undefined otherwise} \end{cases}\\
\text{type}\left(\sum x.E\right) &= \text{type}(E)\\
\text{type}\left(\prod x.E\right) &= \begin{cases} (1,1) \text{ if } \text{type}(E) = (1,1) \\ (n,n) \text{ if } \text{type}(E) = (n,n) \\ \text{undefined} \quad \text{otherwise} \end{cases}
\end{aligned}
$$

We say that $E$ is well typed if it has a defined type. Here, type(undefined)= undefined.

Given $I, \nu, V$ and a well typed expression $E$, we define the semantics for the values $[E]\,(I,\nu)$ inductivelly as follows.

$$
\begin{aligned}
[x]\,(I,\nu) &= \nu(x)\\
[M]\,(I,\nu) &= I(M)\\
[v] &= I(v)\\
[E^*]\,I,\nu) &= [E]\,(I,\nu)^*\\
[E_1 + E_2]\,I,\nu) &= [E_1]\,(I,\nu) + [E_2]\,(I,\nu)\\
[E_1 \cdot E_2]\,I,\nu) &= [E_1]\,(I,\nu) \times [E_2]\,(I,\nu)\\
[f(E_1,\ldots,E_k)]\,I,\nu) &= f\left([E_1]\,(I,\nu),\ldots,[E_k]\,(I,\nu)\right)\\
\left[\sum x.E\right]\,I,\nu) &= \sum_{i=1}^{dim(I)} [E]\,(I,\nu[x \to can(I)[i]])\\
\left[\prod x.E\right]\,I,\nu) &= \prod_{i=1}^{dim(I)} [E]\,(I,\nu[x \to can(I)[i]])
\end{aligned}
$$

Here, $+$ is matrix sum, $\times$ is matrix multiplication and

$$
\begin{aligned}
\sum_{i=1}^{n} E_i &= E_1 + \cdots + E_n\\
\prod_{i=1}^{n} E_i &= E_1 \times \cdots \times E_n
\end{aligned}
$$

Depending on context, we sometimes denote explicitly that an expression $E$ depends on $x$ as $E(x)$.

Note that the definition above is over a set of vector variables $V$. We can naturally extend our language to work with matrix variables as well. Let V be the set of vector a matrix variables, this is $V = \{\{x, y, z, \ldots\}, \{X, Y, Z, \ldots\}\}$. Also $E := X \in V$, type($X$)= $(n, n)$ and $[X](I, \nu) = \nu(X)$.

Now, with matrix variables on the table, we define a new expression

$$\mu X, v.E$$

With

$$\text{type}(\mu X, v.E) = \text{type}(E)$$

and

$$\begin{aligned}[\mu X, v.E](I, \nu) &= A_n \\ A_i &= [E](I, \nu[X \to A_{i-1}, v \to v_i]),\ \ i = 1, \ldots, n \\ A_0 &= Id\end{aligned}$$

Here, $Id$ is the identity. This operator iterates over the canonical vectors and uses the previous result as an input matrix for the same expression

# 4 Iterations in MATLANG

It is natural for us to wish for some kind of iteration in MATLANG. There's a lot of matrix procedures and formulas that sums or multiplies results over an operation on the same matrix. Some examples:

- The quantity
$$\sum_{i=0}^{n} A^i = I + A + A^2 + \ldots + A^n$$
it's used a lot in computing inverse or graph theory. This is a sum over the operation power to the $i$ of the matrix $A$.

- In the LU factorization process the result is
$$A = G_1^{-1} G_2^{-2} \cdots G_n^{-1} U = LU$$
for some upper triangular matrix $U$ and some pivot matrices $G_1, \ldots, G_n$. Note that
$$L = \prod_{i=1}^{n} G_i^{-1}$$

We can compute this and many other things using extMATLANG expressions. The extra property of these expressions relies on the operators $\sum$ and $\prod$.

First of all, note that the definition of these operators is not ambiguous in taking column vectors instead of row vectors or doing the aggregate matrix multiplication by right, because row vectors are the canonical vectors as well, and transposing gives us one from the other. On the other hand, if you want the result of the aggregate multiplication to be computed in the inverse order, note that, since $(AB)^* = B^* A^*$ we have

$$\overset{\text{left}}{\prod} x.E(x) = \left(\prod x.E(x)^*\right)^*.$$

Now, let's see what we can do with these new operators. Some of these examples are simulations of original MATLANG matrix operations. As a consecuence, the semantics of these operators are well defined and we can use them as well. We assume that we have $I, \nu, V$.

### Identity

We can express the identity of dimention $dim(I)$ as

$$\sum x.x \cdot x^*.$$

### Ones vector and diag()

Note that

$$\text{ones}(v) = \sum x.x$$

And

$$\text{diag}(v) = \sum x.(x^* v) \cdot xx^*$$

### Matrix pointwise function application (apply[f])

We have that

- **Pointwise application:** if $A^{(1)}, \ldots, A^{(n)}$ are $m \times p$ matrices, then $\text{apply}[f]\,(A^{(1)}, \ldots, A^{(n)})$ is the $m \times p$ matrix $C$ where $C_{ij} = f(A^{(1)}_{ij}, \ldots, A^{(n)}_{ij})$.

  So, given the **sum** operator and the function $f$, we can compute $C$ in the following way:

  $$C = \sum x_i. \sum x_j.f\left(x_i^* A^{(1)} x_j, \cdots, x_i^* A^{(n)} v_j\right) \cdot x_i x_j^*$$

  Recall that $v_i v_j^*$ is the matrix that has a 1 in the position i, j and zero everywhere else.
  Thus, we can simulate the MATLANG operator apply[$f$].
  For example, matrix pointwise multiplication:

  $$\sum x_i. \sum x_j. \left(x_i^* A^{(1)} x_j \times \cdots \times x_i^* A^{(n)} x_j\right) \cdot x_i x_j^*$$

### Trace and diagonal product

We can express

$$tr(A) = \sum x.x^* Ax.$$

And the product of the diagonal of a matrix (this can be useful in computing the determinant of an upper/down triangular matrix):

$$dp(A) = \prod x.x^* Ax.$$

## Determinant

Recall that the determinant is define over permutations, this is

$$det(A) = \sum_p \sigma(p) a_{1p_1} \cdots a_{np_n},$$

where the sum is taken over the $n!$ permutations of the natural order $(1, 2, \ldots, n)$ and

$$\sigma(p) = \begin{cases} +1 \text{ if p can be restored to natural order in an even number of steps} \\ -1 \text{ if p can be restored to natural order in an odd number of steps} \end{cases}$$

In the case of MATLANG, the permutations are represented as permutation matrices, this is, permutations of the identity. In this context, note that, for a permutation matrix $P$, we compute $a_{1p_1} \cdots a_{np_n}$ as

$$\prod x.x^* \cdot A \cdot (Px).$$

See that $a_{ip_i} = v_i^* \cdot A \cdot (Pv_i)$, where $v_i$ is the $i$-th canonical vector.

We also need to compute $\sigma(P)$, to this end, note that

$$\sigma(P) = (-1)^{\sum_{i<j}\{p_i > p_j\}}.$$

Also, let $\{v_i\}_{i=1}^n$ be the canonical vectors. Now, note that

$$2 \cdot v_i^* Z v_j = \begin{cases} 2 \text{ if } i < j \\ 0 \text{ if } i \geq j \end{cases}$$

Thus

$$1 - 2 \cdot v_i^* Z v_j = \begin{cases} -1 \text{ if } i < j \\ +1 \text{ if } i \geq j \end{cases}$$

So

$$\sigma(P) = \prod x_i . \prod x_j : (i < j) . (1 - 2 \cdot (Px_i)^* Z (Px_j)),$$

and since

$$v_i^* Z v_j = \begin{cases} 1 \text{ if } i < j \\ 0 \text{ if } i \geq j \end{cases}$$

Then we have that

$$\sigma(P) = \prod x_i . \prod x_j . (1 - 2 \cdot (Px_i)^* Z (Px_j) \cdot x_i^* Z x_j)$$

Thus, given we can iterate over permutation matrices, we have

$$det(A) = \sum_P \left[ \left( \prod x_i \prod x_j \, (1 - 2 \cdot (Px_i)^* Z (Px_j) \cdot x_i^* Z x_j) \right) \left( \prod y.y^* \cdot A \cdot (Py) \right) \right].$$

### Transitive closure

To compute transitive closure we need the quantity $(A+I)^n$ and use apply$[f>0]$ on the result matrix. Then we have that

$$(I+A)^n = \prod v.(I+A),$$

and thus

$$TC = \text{apply}\left[f_{>0}\right]\left(\prod v.(I+A)\right).$$

Using only the new operators, we have that

$$TC = \sum v_i.\sum v_j.f_{>0}\left(v_i^*\left(\prod v.(I+A)\right)v_j\right)$$

### Power sum

If we want to compute $I + A + A^2 + \cdots + A^n$ we need to do a little trick with the identity. In this case, we need to add results of matrix powers. We do this like

$$\sum v.\prod w.\left((A-I)(wZv)+I\right).$$

To see why this works, note that

$$wZv = \begin{cases} 1 \text{ if } w < v \\ 0 \text{ if } w \geq v \end{cases}$$

This is, for a sum term $A^k$, if $w$ is previous to $v$ (in the canonical vectors ordering), then stop multiplying $A$ and use $I$ until the end of the current sum term, and then add this term to the final result.

### $k$-cliques

Let's try to see first if there is a four clique in the adjacency matrix $A$. To do this, we need to verify if there are paths between four **different** nodes. So we need the function

$$f(u,v) = 1 - u^*v = \begin{cases} 0 \text{ if } u = v \\ 1 \text{ if } u \neq v \end{cases}$$

Define

$$g(u,v,w,r) = f(u,v)\cdot f(u,w)\cdot f(u,r)\cdot f(v,w)\cdot f(v,r)\cdot f(w,r).$$

This is, $g$ is zero if any pair of vectors are the same.

So

$$\sum v_1.\sum v_2.\sum v_3.\sum v_4.(v_1^*Av_2)(v_1^*Av_3)(v_1^*Av_4)(v_2^*Av_3)(v_2^*Av_4)(v_3^*Av_4)g(v_1,v_2,v_3,v_4).$$

### Gaussian elimination

Let $A$ be a matrix and $\alpha$ a scalar. For computing Gaussian elimination we turn our attention in the elementary matrices of the form

$$E = I + \alpha\cdot e_i e_j^*.$$

Note that $E\cdot A$ adds a $\alpha$-multiple of row $i$ to row $j$. It is worth noting that $E^{-1} = I - \alpha\cdot e_i e_j^*$.

The key property is that, if $A$ is $LU$ factorizable without any row interchange, then $A = LU$, for some upper diagonal matrix $U$ and $L = (E_1 \cdots E_k)^{-1} = E_k^{-1} \cdots E_1^{-1}$ for some elementary matrices $E_i = I + \alpha_i \cdot e_i e_j^*$.

For instance, assume that $A$ has no zero pivots (no row interchanging is necessary), we aim to reduce the first column. Let us do the first reduction, this is, substract a multiple of the first row to the second row. Let $\{v_i\}_i$ be the canonical vectors. We first need to find the proper $\alpha$ to ponderate the first row. In gaussian elimination this is the first entry of the second row divided by the first entry of the first row, this is

$$\alpha = \frac{v_2^* A v_1}{v_1^* A v_1}.$$

So

$$E_1 = I - \alpha_1 \cdot v_1 v_2^*$$

and $E_1 A = A'$ where $A'$ has the second row reduced (first entry zero).

Do this for the rest of the rows and we will have

$$E_k \cdots E_1 A = A'$$

and

$$A'_{i1} = \begin{cases} A_{11} \text{ if } i = 1 \\ 0 \text{ if } i > 1 \end{cases}$$

Note that

$$\begin{aligned} A' &= E_k \cdots E_1 A \\ &= (E_1^* \cdots E_k^*)^* A \\ &= \left( \prod_k E_k^* \right)^* A \\ &= \left( \prod_{v_k \neq v_1} (I - \alpha_k \cdot v_1 v_k^*)^* \right)^* A \\ &= \left( \prod_{v_k \neq v_1} \left( I - \frac{v_k^* A v_1}{v_1^* A v_1} \cdot v_1 v_k^* \right)^* \right)^* A \end{aligned}$$

And we get $A$ with the first column reduced. Note that we need $v_k \neq v_1$ because we don't want to reduce the first row since it will be set to all zeroes. Furthermore, we want to reduce only the downward rows. This can be done with the function

$$f(v_i, v_j) = v_i^* Z v_j = \begin{cases} 1 \text{ if } i < j \\ 0 \text{ if } i \geq j \end{cases}$$

Thus

$$A' = \left( \prod_{v_k} \left( I - \frac{v_k^* A v_1}{v_1^* A v_1} \cdot v_1 v_k^* \cdot (v_1^* Z v_k) \right)^* \right)^* A.$$

So we can reduce the column $i$ of $A$ as

$$\left(\prod_{v_k}\left(I - \frac{v_k^* A v_i}{v_i^* A v_i}\cdot v_i v_k^* \cdot (v_i^* Z v_k)\right)^*\right)^* A.$$

But this is all we can do: reduce a column. For example, we could wrongfully think that all that is left to complete the gaussian elimination on $A = LU$ is to do this for all columns, this is

$$U = \left(\prod_{v_i}\prod_{v_j}\left(I - \frac{v_j^* A v_i}{v_i^* A v_i}\cdot v_i v_j^* \cdot (v_i^* Z v_j)\right)^*\right)^* A.$$

But note that here we obtain the multipliers $\alpha_{ij}$ from the original matrix $A$, a huge mistake. So all we can do with these new operators is to reduce one column of a given matrix. It's not a weak property, but not powerful enough.

## Recursive iteration

Given the above problem, we sort of need an operator that iterates over the canonical vectors and compute the same operation over the result of the previous iteration.

Let $E(X, v)$ be a two variable expression. Recall that

$$\mu X, v.E(X, v)$$

is the operator that iterates over the canonical vectors $\{v_i\}_{i=1}^n$ and successively computes $E(A_{i-1}, v_i)$ where $A_i = E(A_{i-1}, v_i)$ and $A_0 = I$. The value of $\mu X, v.E(X, v)$ is $A_n$. This is

$$\begin{aligned}\mu X, v.E(X, v) &= A_n\\ A_i &= E(A_{i-1}, v_i),\ \ i = 1, \cdots, n\\ A_0 &= I\end{aligned}$$

### LU factorization with recursive iteration

Let's consider the following setting of the $A = LU$ factorization:

$$\begin{aligned}A_i &= T_{i-1} A_{i-1},\ \ i = 2, \cdots, n\\ A_1 &= A\end{aligned}$$

Note that $A_n = U$ and

$$\begin{aligned}A_i &= T_{i-1} A_{i-1}\\ &= \left(\prod_{v_k}\left(I - \frac{v_k^* A_{i-1} v_i}{v_i^* A_{i-1} v_i}\cdot v_i v_k^* \cdot (v_i^* Z v_k)\right)^*\right)^* A_{i-1}.\end{aligned}$$

Now, if $A = LU$ and

$$E(X, v) = \left(\prod_{v_k}\left(I - \frac{v_k^*(XA)v}{v^*(XA)v}\cdot v v_k^* \cdot (v^* Z v_k)\right)^*\right)^* X$$

then $U = (\mu X, v.E(X, v))(A)$, this is $L^{-1} = \mu X, v.E(X, v)$.

Note that in the example above we *inserted* the target matrix $A$ in the expression $E(X, v)$. Would it be different if in the definition of $\mu$ we start with some matrix $B$ and not the indentity? We show that we can simulate this using our setting. Let us have an operator $\lambda$ that does exactly what $\mu$ does, except that we have to give another parameter: a starting matrix. This is:

$$\begin{aligned}\lambda X, v.E(X, v)(B) &= A_n \\ A_i &= E(A_{i-1}, v_i), \;\; i = 1, \cdots, n \\ A_0 &= B\end{aligned}$$

Now, we use the following expression with the operator $\mu$ to simulate $\lambda$:

$$E'(X, v) = min(v) \cdot E(B, v_1) + (1 - min(v))E(X, v).$$

Note that for $\lambda$ we have

$$\begin{aligned}A_0^\lambda &= B \\ A_1^\lambda &= E(A_0^\lambda, v_1) = E(B, v_1) \\ A_2^\lambda &= E(A_1^\lambda, v_2) \\ &\vdots \\ A_n^\lambda &= E(A_{n-1}^\lambda, v_n)\end{aligned}$$

and for $\mu$

$$\begin{aligned}A_0^\mu &= I \\ A_1^\mu &= E'(A_0^\mu, v_1) = E(B, v_1) \\ A_2^\mu &= E'(A_1^\mu, v_2) = E(A_1^\mu, v_2) \\ &\vdots \\ A_n^\mu &= E'(A_{n-1}^\mu, v_n) = E(A_{n-1}^\mu, v_n).\end{aligned}$$

Since $A_1^\mu = A_1^\lambda$ we have that $A_i^\mu = A_i^\lambda$ for $i = 1, \ldots, n$. So

$$\lambda X, v.E(X, v)(B) = \mu X, v.E'(X, v)$$

## Current main questions

- $PA = LU$ factorization: if $A$ needs row interchange, is there some way to compute $P$ beforehand?
  - I am not sure whether you can do this beforehand.
  - But, isn't it possible to simulate *partial pivoting*, i.e., when dealing with canonical vector $v_i$, extract the $i$th column $X[*, i]$ from the current matrix, find the maximal elements in $X[i - n, i]$ and corresponding indicator vector for this (i.e., ones on positions where the max value occurs), then somehow reduce this to a vector in which, say only the smallest occurrence $k$ of ones is retained (smallest position), and then turn this into a permutation matrix that exchanges all rows $X[j, *]$ for $j \geq i$ with the $k$th row, resulting in permuted version of $X$, on which you can then proceed as before for the $i$th step? This will require nested recursion.

- Study if we can express other factorizations ($LDU, LDL$, Cholesky, etc.).
- Starting from $A = LU$, can we compute $A^{-1}$ easily? It's possible that is the gaussian elimination upwards, this is, reduce $A$ to the identity.
- The importance of $Z$ is that it gives us an order. Can we compute $Z$ with the new operators?
- Besides function application, are there any other expressions where the sum operator is useful?
- Can we compute $A^{dim(A)}$?
- Explore the possibility to define the sum and product operators over invariant expressions, and then extend the definition through $Z$.

**Observations**

- Note that if we have
$$f(x) = \begin{cases} 1 \text{ if } P(x) \\ 0 \text{ if } Q(x) \end{cases}$$
If needed, we can compute a piecewise function with any values, like
$$b - (b-a)f(x) = \begin{cases} a \text{ if } P(x) \\ b \text{ if } Q(x) \end{cases}$$

# 5 Connection with logic

We show the expressive power of MATLANG. Let RRA be the class of algebras of binary relations with the operations all, identity, set difference, converse and relational composition. Also, let $\text{FO}_b^3$ denote first-order logic with three variables, equality and infinitely many binary relation symbols. This is, $\text{FO}_b^3$ are $\text{FO}^3$ graph queries.

It is known that the logic captured by RRA is $\text{FO}_b^3$. Now, we show that RRA can be interpreted into MATLANG, thus the expressive power of MATLANG is at least the same as $\text{FO}_b^3$.

Let $U$ be a nonempty finite set with $n$ elements (and thus has an enumeration $u_1, \ldots, u_n$). Let $\mathcal{A}^U \in$ RRA, this is
$$\mathcal{A}^U = \langle A, \cup, -, \circ, ^{-1}, I \rangle,$$
where

- $A \subseteq \mathcal{P}(U \times U)$, this is, $\forall R \in A. R \subseteq U \times U$. So $A$ is a set of binary relations over $U$.
- $\cup, -, \circ, ^{-1}$ denote the operations union, set difference, relational composition and converse, respectively. All of them defined over binary relations.
- $I$ is the constant relation symbol that denotes the set $\{(u, u) : u \in U\}$.

Let $R \in A$. Define $M^R$ a matrix such that
$$M_{ij}^R = \begin{cases} 1 & \text{if } (u_i, u_j) \in R \\ 0 & \text{if } (u_i, u_j) \notin R \end{cases}$$

Note that the MATLANG instance of $\mathcal{A}$, has $|A|$ matrices with dimentions $|U| \times |U|$. Thus binary relations are represented as adyacency matrices.

We know show how to express the RRA operations in MATLANG.

- **All:** let $R \in A$ be any relation. We express $U \times U$ as $M^{U\times U} = \mathbf{1}(M^R) \cdot \mathbf{1}(M^R)^*$.
- **Identity:** let $R \in A$ be any relation. Then we express the constant relation $I$ as $M^I = \text{diag}(\mathbf{1}(M^R))$.
- **Union:** let $R, S \in A$. Then $M^{R\cup S} = \text{apply}[x \vee y](M^R, M^S)$.
- **Set difference:** let $R, S \in A$. Then $M^{R-S} = \text{apply}[x \vee \neg y](M^R, M^S)$.
- **Converse:** let $R \in A$. Then we express $R^{-1} = \{(u, v) \in U \times U : (v, u) \in R\}$ as $M^{R^{-1}} = (M^R)^*$.
- **Relational composition:** let $R, S \in A$, then $M^{R\circ S} = \text{apply}[x > 0](M^R \cdot M^S)$. Where

$$x > 0 = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

Thus RRA can be interpreted into MATLANG, and hence so does $\text{FO}^3_b$, this is, $\text{FO}^3$ graph queries.

# 6 Connection with logic with aggregates

We first recall how MATLANG expressions can be simulated in the calculus with aggregates (see Expressive power of SQL paper by Leonid for definition of this calculus). I am copying stuff from the ICDT paper (and upcoming TODS submission).

To fix the relational representation of matrices, it is natural to represent an $m \times n$ matrix $A$ by a ternary relation

$$\mathit{Rel}_2(A) := \{(i, j, A_{i,j}) \mid i \in \{1, \ldots, m\},\ j \in \{1, \ldots, n\}\}.$$

In the special case where $A$ is an $m \times 1$ matrix (column vector), $A$ can also be represented by a binary relation $\mathit{Rel}_1(A) := \{(i, A_{i,1}) \mid i \in \{1, \ldots, m\}\}$. Similarly, a $1 \times n$ matrix (row vector) $A$ can be represented by $\mathit{Rel}_1(A) := \{(j, A_{1,j}) \mid j \in \{1, \ldots, n\}\}$. Finally, a $1 \times 1$ matrix (scalar) $A$ can be represented by the unary singleton relation $\mathit{Rel}_0(A) := \{(A_{1,1})\}$.

More formally, we assume a supply of *relation variables*, which, for convenience, we can take to be the same as the matrix variables. A *relation type* is a tuple of **b**'s and **n**'s. A *relational schema* $\mathcal{S}$ is a function, defined on a nonempty finite set $\mathit{var}(\mathcal{S})$ of relation variables, that assigns a relation type to each element of $\mathit{var}(\mathcal{S})$.

To define relational instances, we assume a countably infinite universe **dom** of abstract atomic data elements. It is convenient to assume that the natural numbers are contained in **dom**. We stress that this assumption is not essential but simplifies the presentation. Alternatively, we would have to work with explicit embeddings from the natural numbers into **dom**.

Let $\tau$ be a relation type. A *tuple of type* $\tau$ is a tuple $(t(1), \ldots, t(n))$ of the same arity as $\tau$, such that $t(i) \in \mathbf{dom}$ when $\tau(i) = \mathbf{b}$, and $t(i)$ is a complex number when $\tau(i) = \mathbf{n}$. A *relation of type* $\tau$ is a finite set of tuples of type $\tau$. An *instance* of a relational schema $\mathcal{S}$ is a function $I$ defined on $\mathit{var}(\mathcal{S})$ so that $I(R)$ is a relation of type $\mathcal{S}(R)$ for every $R \in \mathit{var}(\mathcal{S})$.

The matrix data model can now be formally connected to the relational data model, as follows. Let $\tau = s_1 \times s_2$ be a matrix type. Let us call $\tau$ a *general type* if $s_1$ and $s_2$ are both size symbols; a *vector type* if $s_1$ is a size symbol and $s_2$ is 1, or vice versa; and the *scalar type* if $\tau$ is $1 \times 1$. To every matrix type $\tau$ we associate a relation type

$$Rel(\tau) := \begin{cases} (\mathbf{b}, \mathbf{b}, \mathbf{n}) & \text{if } \tau \text{ is general;} \\ (\mathbf{b}, \mathbf{n}) & \text{if } \tau \text{ is a vector type;} \\ (\mathbf{n}) & \text{if } \tau \text{ is scalar.} \end{cases}$$

Then to every matrix schema $\mathcal{S}$ we associate the relational schema $Rel(\mathcal{S})$ where $Rel(\mathcal{S})(M) = Rel(\mathcal{S}(M))$ for every $M \in var(\mathcal{S})$. For each instance $I$ of $\mathcal{S}$, we define the instance $Rel(I)$ over $Rel(\mathcal{S})$ by

$$Rel(I)(M) = \begin{cases} Rel_2(I(M)) & \text{if } \mathcal{S}(M) \text{ is a general type;} \\ Rel_1(I(M)) & \text{if } \mathcal{S}(M) \text{ is a vector type;} \\ Rel_0(I(M)) & \text{if } \mathcal{S}(M) \text{ is the scalar type.} \end{cases}$$

**Proposition 1.** *Let $\mathcal{S}$ be a matrix schema, and let $e$ be a* MATLANG *expression that is well-typed over $\mathcal{S}$ with output type $\tau$. Let $\ell = 2$, 1, or 0, depending on whether $\tau$ is general, a vector type, or scalar, respectively. For every* MATLANG *expression $e$ there is a formula $\varphi_e$ over $Rel(\mathcal{S})$ in the relational calculus with summation, such that*

1. *If $\tau$ is general, $\varphi_e(i, j, z)$ has two free base variables $i$ and $j$ and one free numerical variable $z$; if $\tau$ is a vector type, we have $\varphi_e(i, z)$; and if $\tau$ is scalar, we have $\varphi_e(z)$.*
2. *For every instance $I$, the relation defined by $\varphi_e$ on $Rel(I)$ equals $Rel_\ell(e(I))$.*
3. *The formula $\varphi_e$ uses only* **three distinct base variables**. *The functions used in pointwise applications in $\varphi_e$ are those used in pointwise applications in $e$; complex conjugation; multiplication of two numbers; and the constant functions 0 and 1. Furthermore, $\varphi_e$ neither uses equality conditions between numerical variables nor equality conditions on base variables involving constants.*

Let us assign, to each MATLANG expression $e$ that is well-typed over $\mathcal{S}$, an expression $\varphi_e$ in the relational calculus with summation as follows.

- If $e = M$ is a matrix variable of $\mathcal{S}$, then $\varphi_e(i, j, x) := Rel_2(M)(i, j, x)$ if $M$ is of general type, $\varphi_e(i, x) := Rel_1(M)(i, x)$ if $M$ is of vector type, and $\varphi_e(x) := Rel_0(M)(x)$ if $M$ is of scalar type.

Let $e'$ be a MATLANG and let $\tau = s_1 \times s_2$ be the output type of $e'$.

- If $e = (e')^*$, then $\varphi_e(i, j, x) := \exists x' \left(\varphi_{e'}(j, i, x') \wedge x = \overline{x'}\right)$ if $\tau$ is a general type, $\varphi_e(i, x) := \exists x' \left(\varphi_{e'}(i, x') \wedge x = \overline{x'}\right)$ if $\tau$ is a vector type, and $\varphi_e(x) := \exists x' \left(\varphi_{e'}(x') \wedge x = \overline{x'}\right)$ if $\tau$ is the scalar type. Here, $\overline{x}$ denotes the complex conjugate operation.
- If $e = \mathbf{1}(e')$, then $\varphi_e(i, x) := \exists j, x' \left(\varphi_{e'}(i, j, x') \wedge x = 1(x')\right)$ if $\tau$ is a general type, $\varphi_e(i, x) := \exists x' \left(\varphi_{e'}(i, x') \wedge x = 1(x')\right)$ is a vector type and $s_1 \neq 1 = s_2$, $\varphi_e(x) := \exists i, x' \left(\varphi_{e'}(i, x') \wedge x = 1(x')\right)$ is a vector type and $s_1 = 1 \neq s_2$, and $\varphi_e(x) := \exists x' \left(\varphi_{e'}(x') \wedge x = 1(x')\right)$ if $\tau$ is the scalar type. As before, 1 in the expression $\varphi_e$ is the constant 1 function.

- If $e = \mathrm{diag}(e')$, then $\varphi_e(i,j,x) := (\varphi_{e'}(i,x) \wedge j = i) \vee (\exists x', x''\, \varphi_{e'}(i,x') \wedge \varphi_{e'}(j,x'') \wedge i \neq j \wedge x = 0(x')$ if $s_1 \neq 1 = s_2$ and $\varphi_e(x) := \varphi_{e'}(x)$ if $\tau$ is the scalar type.

- If $e = e_1 \cdot e_2$, then

$$\begin{cases} \varphi_e(i,j,z) := z = \mathsf{sum}k, x, y.(\varphi_{e_1}(i,k,x) \wedge \varphi_{e_2}(k,j,y), x \times y) & \text{if } s_1 \neq 1 \neq s_2 \text{ and } s_3 \neq 1 \\ \varphi_e(i,z) := z = \mathsf{sum}k, x, y.(\varphi_{e_1}(i,k,x) \wedge \varphi_{e_2}(k,y), x \times y) & \text{if } s_1 \neq 1 = s_2 \text{ and } s_3 \neq 1 \\ \varphi_e(i,z) := z = \mathsf{sum}k, x, y.(\varphi_{e_1}(k,x) \wedge \varphi_{e_2}(k,i,y), x \times y) & \text{if } s_1 = 1 \neq s_2 \text{ and } s_3 \neq 1 \\ \varphi_e(z) := z = \mathsf{sum}k, x, y.(\varphi_{e_1}(k,x) \wedge \varphi_{e_2}(k,y), x \times y) & \text{if } s_1 = 1 = s_2 \text{ and } s_3 \neq 1 \\ \varphi_e(i,j,z) := \varphi_{e_1}(i,x) \wedge \varphi_{e_2}(j,y) \wedge z = x \times y & \text{if } s_1 \neq 1 \neq s_2 \text{ and } s_3 = 1 \\ \varphi_e(i,z) := \varphi_{e_1}(i,x) \wedge \varphi_{e_2}(y) \wedge z = x \times y & \text{if } s_1 \neq 1 = s_2 \text{ and } s_3 = 1 \\ \varphi_e(i,z) := \varphi_{e_1}(x) \wedge \varphi_{e_2}(i,y) \wedge z = x \times y & \text{if } s_1 = 1 \neq s_2 \text{ and } s_3 = 1 \\ \varphi_e(z) := \varphi_{e_1}(x) \wedge \varphi_{e_2}(y) \wedge z = x \times y & \text{if } s_1 = 1 = s_2 \text{ and } s_3 = 1 \end{cases},$$

  where $e_1$ is of type $s_1 \times s_3$ and $e_2$ is of type $s_3 \times s_2$.

- If $e = \mathsf{apply}[f](e_1, \ldots, e_n)$, then

$$\begin{aligned} \varphi_e(i,j,x) &:= \exists x_1, \ldots, x_n\, (\varphi_{e_1}(i,j,x_1) \wedge \cdots \wedge \varphi_{e_n}(i,j,x_n) \wedge x = f(x_1, \ldots, x_n)), \\ \varphi_e(i,x) &:= \exists x_1, \ldots, x_n\, (\varphi_{e_1}(i,x_1) \wedge \cdots \wedge \varphi_{e_n}(i,x_n) \wedge x = f(x_1, \ldots, x_n)), \text{ and} \\ \varphi_e(x) &:= \exists x_1, \ldots, x_n\, (\varphi_{e_1}(x_1) \wedge \cdots \wedge \varphi_{e_n}(x_n) \wedge x = f(x_1, \ldots, x_n)) \end{aligned}$$

  depending on whether $\tau$ is of general, vector or scalar type, respectively.

Notice that the only functions in $\varphi_e$ aside from those used in $\mathsf{apply}$ in $e$ are complex conjugation ($\bar{z}$), multiplication of two numbers ($\times$), and the constant functions 0 and 1. Also notice that $\varphi_e$ uses neither negation, nor equality conditions on numerical variables, nor equality conditions on variables involving a constant.

By induction on the structure of $e$ one straightforwardly observes that $\varphi_e$ satisfies the conditions (1) and (2) in the statement of the theorem. Furthermore, it is clear for all operations except for matrix multiplication that when $\varphi_{e'}$ uses at most 3 base variables than so does $\varphi_e$. When it comes to matrix multiplication, assume that $\varphi_{e_1}(i,k,x)$ uses base variables $i, j', k$ and $\varphi_{e_2}(k,j,y)$ use base variables $i', k, j$. Then since only $i$, $j$ and $k$ are free variables, we can simply reuse variable $j'$ for $i'$ (or vice versa). Hence, $\varphi_e(i,j,z) := z = \mathsf{sum}k, x, y.(\varphi_{e_1}(i,k,x) \wedge \varphi_{e_2}(k,j,y), x \times y)$ uses at most 3 base variables as well. □

### 6.1 MATLANG + sum iteration

Let's denote this extension of MATLANG by $\sum$MATLANG. We show that we can extend the previous translation to incorporate expressions of the form $\sum_v e(v)$. More precisely, let $e$ be a $\sum$MATLANG expression. Such an expression $e$ is defined on matrix variables which hold to input matrices. To formally define $\sum_v e(v)$ we reserve an infinite set of vector variables $V_1, V_2, \ldots$, which are to hold the canonical vectors over which the sum ranges. So, in general, we consider expressions of the form

$$\sum_v e(V_1/v, V_2, \ldots, V_k),$$

where $V_1/v$ means that vector variable is instantiated by vector $v$, the other vector variables $V_2, \ldots, V_k$ are "free" (a notion we need to define for $\sum$MATLANG expressions) and these will

be instantiated later by further summation (i.e., in the end a valid $\sum$MATLANG expression does not have free vector variables).

It is easy to see that the inductive proof given above still works in the presence of such vector variables. The difference is that $\varphi_e$ will have occurrences of $Rel_1(V_j)(i,x)$ which, when considering an instance $I$ of the matrix variables the vector variables will be instantiated by means of the sum iteration. As a consequence, for every such occurrence of $Rel_1(V_j)(i,x)$ we have an extra pair of free variables (one of **b** type for the index $i$ in the vector, and one of **n** type for the value $x$ of the entries in the vector. In summary, $\varphi_e$ is of the form $\varphi_e(i,j,x;i_1,x_1,\ldots,i_k,x_k)$ where the last $2k$ variables correspond to the vector variables.

Consider now

$$e := \sum_v e'(V_1/v, V_2, \ldots, V_k),$$

we assume that $e'$ is of matrix type (if $e'$ is of vector type or scalar we may need to encode these differently, shouldn't be a problem).

Then to translate $e$ into the calculus we

- first consider $\varphi_{\mathsf{Id}}(i,j,x)$ as the expression corresponding to $\mathsf{diag}(\mathbf{1}(M))$ for a matrix variable $M$, this is to generate the identity matrix.
- Then, we replace the vector relation $Rel_1(V_1)(i_1,x_1)$ in $\varphi'_e(i,j,x;i_1,x_1,\ldots,i_k,x_k)$ by columns returned by $\varphi_{\mathsf{Id}}(i_1,j',x)$ for a given $j'$:

  $$\psi_e(i,j,x;j',i_2,x_2,\ldots,i_k,x_k) := \exists i_1,x_1\, \varphi_{\mathsf{Id}}(i_1,j',x_1) \wedge \varphi_{e'}(i,j,x;i_1,x_1,\ldots,i_k,x_k)[V_1(i_1,x_1)/\varphi_{\mathsf{Id}}(i_1,j',x_1)],$$

  where $\varphi_{e'}(i,j,x;i_1,x_1,\ldots,i_k,x_k)[V_1(i_1,x_1)/\varphi_{\mathsf{Id}}(i_1,j',x_1)]$ means that we subtitute $V_1(i_1,x_1)$ by $\varphi_{\mathsf{Id}}(i_1,j',x_1)$ everywhere in expression $\varphi_{e'}$. We further existentially quantify $i_1$ and $x_1$ so that we when $\psi_e$ is evaluated on an instance $I$ (for matrix variables) and index $j$ (and indexes $i_2,\ldots,i_k$ and values $x_2,\ldots,x_k$, $\psi_e$ evaluates $\varphi_{e'}$ were $V_1$ holds the $j$th column of the identity matrix.
- It remains to sum up over all these columns in the identity matrix:

  $$\varphi_e(i,j,z;i_2,x_2,\ldots,i_k,x_k) := z = \mathsf{sum}x,j'(\exists i',x',\, \varphi_{\mathsf{Id}}(i',j',x') \wedge \psi_e(i,j,x;j',i_2,x_2,\ldots,i_k,x_k),x)$$

  which results in an expression in which the pair of variables for $V_1$ is eliminated.

## 6.2 Conjecture

Under certain syntactical restriction on function applications and equality conditions (no numerical equal to base variable) it is possible to translate any calculus with sum expression which starts from relations encoding matrices and has an output type that correspond to a scalar, vector or matrix, into $\sum$MATLANG? Perhaps also no negation in the expression? The challenge will be to simulate conjunction (disjunction), in other words. For examples, we can have sub-formulas $M(i,j,x) \wedge M(i',j',x')$ which will be existentially quantified in the end because we can only have three variables (indexes $i$ and $j$ and numerical variable) in the final expression.

Also note that we only need to consider expressions that are "functional". For example,

$$\varphi(i,j',z) = \exists j,i'(R(i,j,x) \wedge R(i',j',y) \wedge z = f(x,y))$$

for some function $f$ is not "functional" in the sense that for a given $i$ and $j'$ there may be more than one value $z$. Any inductive translation will need to require functionally for all sub-formulas

which may help. On the other hand, an expression may be functional while not all sub-queries are so. Indeed, one can always guard a formula with an expression which be true if and only it is functional (just express that there are no two values corresponding to the same entry) and return say the zero matrix if it is not functional. We can turn any formula in a functional one in this way.

More generally, suppose that input relations have keys, what is the complexity of deciding whether the output to a query also satisfies a specified key. Probably undecidable if we have negation but what about positive fragments possibly with aggregates? This should have been studied.

# 7 From Product to Sum

A quick note how product iterations and sum iterations relate. Consider the computation $(I+A)^n$ which we can express using product iterations

$$(I+A)^n = \prod_{e_i}(I+A),$$

but not with sum iterations (as MATLANG+sum can be encoded in logic with aggregation, which is local, and we can define TC with $(I+A)^n$). There is a way, however, of encoding $(I+A)^n$ as sum by extending MATLANG with some extra (powerful) machinery. Intuitively, the $i$th iteration in the product will be stored in the *$i$-slice* $B[i,\_,\_]$ of a $n \times n \times n$ array $B$ and in each sum iteration we additively populate the $i+1$-slice using the result two other slices, the 1-slice $B[1,\_,\_]$ storing $I+A$, and the $i$-slice $B[i,\_,\_]$ storing the result of the previous iteration. So if we consider

$$B[i,j,k] = B[1,j,\ell] \cdot B[i-1,\ell,k]$$

we compute $(I+A)^i$ in $B[i,\_,\_]$ when initially $B[1,\_,\_] := I+A$. Intuitively, we built-in some kind of recursion.

To formalize this in terms of an extension of MATLANG, however, we need to introduce various operations which enable to:

- Store a matrix in a specific slice of a ternary array;
- Extract matrix in a specific slice of a ternary array;
- Extend product to array/matrix, array/vector, and so; **and**
- we need to be able to take the predecessor of a canonical vector (to simulate accessing $i-1$ when given $i$).

This will definitely go beyond first-order logic with aggregation but may be embeddable in some datalog with aggregation variant (with predecessor relation).

The **main reason** for doing this may be motivated by some paper that I came across in which the communication cost of linear algebra operations is studied (in the context of implementations in which data needs to be shipped from slow to fast memory). What is nice about this work is that they provide a general mechanism of obtaining *lower bounds* by using techniques like Loomis-Whitney (also used in AGM work on joins). They develop an entire framework for bounding complexity when programs are **declaratively specified as for loops**!. They also show that the lower bounds are attainable by concrete algorithms (basically by partitioning the indices used in for loops). We could attempt to ensure that the MATLANG extension we consider can only express for loops to which their theory is applicable. The pape r is:

Communication Lower Bounds and Optimal Algorithms for Programs That Reference Arrays - Part 1, by Michael Christ James Demmel Nicholas Knight Thomas Scanlon Katherine A. Yelick. `https://www2.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-61.pdf`.

A colleague working in that area told me that this is state-of-the art!

# 8 Boolean MATLANG

Since we want to do queries over graphs with MATLANG. it is worth to consider the boolean case, this is, when the matrix entries are 1 or 0.

For this purpose, we need to redefine the basic operations of MATLANG if necessary. Note that the only operations that need to be redefined are the matrix multiplication and function application, since the **transpose**, **one-vector** and **diag** of a boolean matrix or vector are boolean matrices or vectors.

- **Matrix multiplication:** if $A$ is a $n \times m$ matrix and $B$ is a $m \times p$ matrix then $A \cdot B$ is a $n \times p$ matrix where $(A \cdot B)_{ij} = \bigvee_{k=1}^{n} A_{ik} \wedge B_{kj}$.
- **Pointwise application:** f $A^{(1)}, \ldots, A^{(n)}$ are $m \times p$ matrices, then apply$[f]\,(A^{(1)}, \ldots, A^{(n)})$ is the $m \times p$ matrix $C$ where $C_{ij} = f(A_{ij}^{(1)}, \ldots, A_{ij}^{(n)})$.

  Note that in the boolean case we need $f : \{0,1\} \to \{0,1\}$. So, for a fixed $n$ there are finitely many functions to apply: $2^{2^n}$. Let's call this set $\mathcal{F}_n$. This is $|\mathcal{F}_n| = 2^{2^n}$.

  Since we are in the boolean case, the following condition holds.

$$\forall f \in \mathcal{F}_n \exists g \in \mathcal{F}_n : f(x_1, \ldots, x_n) = \neg g(x_1, \ldots, x_n),$$

  so we can simulate all $2^{2^n}$ functions with a subset of $\mathcal{F}_n$ of $2^{2^n - 1}$ functions.

# 9 Expressions as functions

Another way of looking at expressions in MATLANG is as functions between matrix spaces, this is

$$e : (M_1, \ldots, M_k) \to M,$$

where $M, M_1, \ldots, M_k$ are matrix spaces, i.e., $M, M_1, \ldots, M_k \in \{\mathcal{M}^{n \times m} : n, m \in \mathbb{N}^+\}$.

Some examples:

- If $A$ is $n \times m$ then

$$\begin{aligned} e(A) = t(A) : \mathcal{M}_{n \times m} &\to \mathcal{M}_{m \times n} \\ A &\to A^* \end{aligned}$$

- If $A$ is $n \times m$ then

$$e(A) = \mathbf{1}(A) : \mathcal{M}_{n\times m} \to \mathcal{M}_{n\times 1}$$
$$A \to n \text{ times} \begin{cases} \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} \\ \begin{bmatrix} \vdots \\ 1 \end{bmatrix} \end{cases}$$

- If $v$ is $n \times 1$ then

$$e(v) = \text{diag}(v) : \mathcal{M}_{n\times 1} \to \mathcal{M}_{n\times n}$$
$$v \to \begin{bmatrix} v_1 & 0 & 0 & \dots & 0 \\ 0 & v_2 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & v_n \end{bmatrix}$$

- If $A$ is $n \times m$ and $B$ is $m \times p$ then

$$e(A, B) = A \cdot B : \mathcal{M}_{n\times m} \times \mathcal{M}_{m\times p} \to \mathcal{M}_{n\times p}$$
$$(A, B) \to A \cdot B$$

- If $A^{(1)}, \dots, A^{(n)}$ are $m \times p$ matrices then

$$e(A^{(1)}, \dots, A^{(n)}) = \text{apply}[f](A^{(1)}, \dots, A^{(n)})$$

  has domains

$$\mathcal{M}_{m\times p}^{n} \to \mathcal{M}_{m\times p}$$
$$(A^{(1)}, \dots, A^{(n)}) \to C : C_{ij} = f(A_{ij}^{(1)}, \dots, A_{ij}^{(n)}).$$

We can start to analyze if this functions are increasing, decreasing, boolean, etc. Also, we can study the effects of disturbances on the input in the output of these functions.

# 10 Core of Matlab and R

## MATLAB

The basic operations of MATLAB over matrices are:

- **mldivide**(A, B): returns $x$ such that $Ax = B$.
- **descomposition**(A): returns a decomposition or factorization $LU, LDL, QR$, Cholesky, etc.
- **inv**(A): returns $A^{-1}$.
- **multiplication**: compute $A \cdot B$.

- **transpose**(A): returns $A^T$.
- **conjugate transpose**(A): returns $A'$.
- **matrix power**(A, k): returns $A^k$.
- **eigen**(A): returns the eigenvectors and the eigenvectors matrices of $A$.
- **funm**(A, f): returns matrix $B$ with elements $b_{ij} = f(a_{ij})$.
- **crossprod**(a, b): vectorial product, returns $c$ such that $c \perp a, b$.
- **dotprod**(a,b): returns $a \cdot b$.
- **diag**(v): $v$ vector. Returns the following matrix:

$$\begin{bmatrix} v_1 & 0 & 0 & \dots & 0 \\ 0 & v_2 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & v_n \end{bmatrix}$$

- **diag**(A): given matrix $A$, it returns

$$\begin{bmatrix} a_{11} \\ a_{22} \\ \vdots \\ a_{nn} \end{bmatrix}$$

- **det**(A): returns the determinant of $A$.
- **zeros**(n, m): returns a $n \times m$ matrix full of zeros.
- **ones**(n, m): returns a $n \times m$ matrix full of ones.
- **A[i, j]**: you can get $A_{ij}$.

### R

The basic operations of the language R over matrices are:

- **A%*%B**: matrix multiplication.
- **A*B**: pointwise multiplication.
- **t(A)**: transpose.
- **diag(v)**: returns the matrix

$$\begin{bmatrix} v_1 & 0 & 0 & \dots & 0 \\ 0 & v_2 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & v_n \end{bmatrix}$$

- **diag(A)**: Returns the vector
$$\begin{bmatrix} a_{11} \\ a_{22} \\ \vdots \\ a_{nn} \end{bmatrix}$$
- **diag(k)**: $k$ scalar. It creates the $k \times k$ identity matrix.
- **matrix(k, n, m)**: returns the $n \times m$ matrix, where every entry is equal to $k$.
- **solve(A, b)**: returns $x$ such that $Ax = b$.
- **solve(A)**: returns $A^{-1}$.
- **det(A)**: determinant of $A$.
- **y<-eigen(A)**: stores de eigenvalues of $A$ in y$val and the eigenvectors in y$vec.
- **y<-svd(A)**: it computes and stores the following:
  - y$d: vector of the singular values of $A$.
  - y$u: matrix of the left singular vectors of $A$.
  - y$v: matrix of the right singular vectors of $A$.
- **R<-chol(A)**: Cholesky fatorization, $R'R = A$.
- **y<-qr(A)**: $QR$ decomposition, strored in y$qr.
- **cbind(A,B, v, ...)**: joins matrices and vector horizontally, returns a matrix.
- **rbind(A,B, v, ...)**: joins matrices and vector vertically, returns a matrix.
- **rowMeans(A)**: returns the vector of the averages over the rows of $A$.
- **colMeans(A)**: returns the vector of the averages over the columns of $A$.
- **rowSums(A)**: returns the vector of the sums over the rows of $A$.
- **colSums(A)**: returns the vector of the sums over the columns of $A$.
- **outer(A, B, f)**: applies $f(\cdot, \cdot)$. Returns matrix $C$ of entries $c_{ij} = f(a_{ij}, b_{ij})$.
- **A[i, j]**: you can get $A_{ij}$.