

# 1 plml: Prolog-Matlab interface

**To be done** Use `mat(I)` and `tmp(I)` as types to include engine Id.

Clarify relationship between return values and valid Matlab denotation.

Reshape/2 array representation: `reshape([ ... ],Size)` Expression language:  
`arr(Vals,Shape,InnerFunctor)` - allows efficient representation of arrays of arbitrary things. Will  
 require more strict nested list form.

Deprecate old `array(Vals::Type)` and `cell(Vals::Type)` left-value syntax.

Remove `I` from `ml_expr//2` and add to `mx` type?

**Types** `ml_eng` - Any atom identifying a Matlab engine.

`ml_stmt` - A Matlab statement

```
X;Y      :: ml_stmt :- X:ml_stmt, Y:ml_stmt.
X,Y      :: ml_stmt :- X:ml_stmt, Y:ml_stmt.
X=Y      :: ml_stmt :- X:ml_lval, Y:ml_expr.
hide(X)  :: ml_stmt :- X:ml_stmt.
```

```
ml_expr(A)      % A Matlab expression, possibly with multiple return values
ml_loc ---> mat(atom,atom). % Matbase locator
```

**Matlab expression syntax** The Matlab expression syntax adopted by this module allows Prolog terms to represent or denote Matlab expressions. Let  $T$  be the domain of recognised Prolog terms (corresponding to the type `ml_expr`), and  $M$  be the domain of Matlab expressions written in Matlab syntax. Then  $V : T \rightarrow M$  is the valuation function which maps Prolog term  $X$  to Matlab expression  $V[X]$ . These are some of the constructs it recognises:

Constructs valid only in top level statements, not subexpressions:

```
X;Y      % |--> V[X]; V[Y] (sequential evaluation hiding first result)
X,Y      % |--> V[X], V[Y] (sequential evaluation displaying first result)
X=Y      % |--> V[X]=V[Y] (assignment, X must denote a valid left-value)
hide(X)  % |--> V[X]; (execute X but hide return value)
```

Things that look and work like Matlab syntax (more or less):

```
+X      % |--> uplus(V[X])
-X      % |--> uminus(V[X])
X+Y     % |--> plus(V[X],V[Y])
X-Y     % |--> minus(V[X],V[Y])
X^Y     % |--> mpower(V[X],V[Y])
X*Y     % |--> mtimes(V[X],V[Y])
X/Y     % |--> mrdivide(V[X],V[Y])
X\Y     % |--> mldivide(V[X],V[Y])
X.^Y    % |--> power(V[X],V[Y])
X.*Y    % |--> times(V[X],V[Y])
X./Y    % |--> rdivide(V[X],V[Y])
X.\Y    % |--> ldivide(V[X],V[Y])
X:Y:Z   % |--> colon(V[X],V[Y],V[Z])
X:Z     % |--> colon(V[X],V[Z])
X>Z     % |--> gt(V[X],V[Y])
X>=Z    % |--> ge(V[X],V[Y])
X<Z     % |--> lt(V[X],V[Y])
X<=Z    % |--> le(V[X],V[Y])
```

|                          |   |
|--------------------------|---|
| <code>X==Z</code>        | %  --> <code>eq(V[X],V[Y])</code>           |
| <code>[X1,X2,...]</code> | %  --> <code>[ V[X1], V[X2], ... ]</code>   |
| <code>[X1;X2;...]</code> | %  --> <code>[ V[X1]; V[X2]; ... ]</code>   |
| <code>{X1,X2,...}</code> | %  --> <code>{ V[X1], V[X2], ... }</code>   |
| <code>{X1;X2;...}</code> | %  --> <code>{ V[X1]; V[X2]; ... }</code>   |
| <code>@X</code>          | %  --> <code>@V[X]</code> (function handle) |

Things that do not look like Matlab syntax but provide standard Matlab features:

|                          |  |
|--------------------------|--|
| <code>'Infinity'</code>  | %  --> <code>inf</code> (positive infinity)                                    |
| <code>'Nan'</code>       | %  --> <code>nan</code> (not a number)   |
| <code>X'</code>          | %  --> <code>ctranpose(V[X])</code> (conjugate transpose, <code>V[X]'</code> ) |
| <code>X#Y</code>         | %  --> <code>getfield(V[X],V[q(Y)])</code>                                     |
| <code>X\\Y</code>        | %  --> <code>@(V[X])V[Y]</code> (same as <code>lambda(X,Y)</code> )            |
| <code>\\Y</code>         | %  --> <code>@()V[Y]</code> (same as <code>thunk(Y)</code> )                   |
| <code>lambda(X,Y)</code> | %  --> <code>@(V[X])V[Y]</code> (anonymous function with arguments X)          |
| <code>thunk(Y)</code>    | %  --> <code>@()V[Y]</code> (anonymous function with no arguments)             |
| <code>vector(X)</code>   | %  --> <code>horzcat(V[X1],V[X2], ...)</code>                                  |
| <code>atvector(X)</code> | % as vector but assumes elements of X are assumed all atomic                   |
| <code>cell(X)</code>     | % construct 1xN cell array from elements of X                                  |
| <code>'X</code>          | % same as <code>q(X)</code>  |
| <code>q(X)</code>        | % wrap <code>V[X]</code> in single quotes (escaping internal quotes)           |
| <code>qq(X)</code>       | % wrap <code>V[X]</code> in double quotes (escaping internal double quotes)    |
| <code>tq(X)</code>       | % wrap TeX expression in single quotes (escape internal quotes)                |

Referencing different value representations.

|                               |  |
|-------------------------------|--|
| <code>mat(X,Y)</code>         | % denotes a value in the Matbase using a dbload expression |
| <code>mx(X:mx_blob)</code>    | % denotes an MX Matlab array in SWI memory                 |
| <code>ws(X:ws_blob)</code>    | % denotes a variable in a Matlab workspace                 |
| <code>wsseq(X:ws_blob)</code> | % workspace variable containing list as cell array.        |

Tricky bits.

|                              |   |
|------------------------------|---|
| <code>apply(X,AX)</code>     | % X must denote a function or array, applied to list of arguments AX. |
| <code>cref(X,Y)</code>       | % cell dereference,  --> <code>V[X]{ V[Y1], V[Y2], ... }</code>       |
| <code>arr(Lists)</code>      | % multidimensional array from nested lists.                           |
| <code>arr(Lists,Dims)</code> | % multidimensional array from nested lists.                           |

Things to bypass default formatting

|                         |   |
|-------------------------|---|
| <code>noeval(_)</code>  | % triggers a failure when processed   |
| <code>atom(X)</code>    | % write atom X as <code>write/1</code>  |
| <code>term(X)</code>    | % write term X as <code>write/1</code>  |
| <code>\(P)</code>       | % escape and call phrase P directly to generate Matlab string                                     |
| <code>\$(X)</code>      | % calls <code>pl2ml_hook/2</code> , denotes <code>V[Y]</code> where <code>plml_hook(X,Y)</code> . |
| <code>'\$VAR'(N)</code> | % gets formatted as <code>p_N</code> where N is assumed to be atomic.                             |

All other Prolog atoms are written using `write/1`, while other Prolog terms are assumed to be calls to Matlab functions named according to the head functor. Thus `V[ <head>( <arg1>, <arg2>, ... ) ] = <head>(V[<arg1>, V[<arg2>], ...)`.

There are some incompatibilities between Matlab syntax and Prolog syntax, that is, syntactic structures that Prolog cannot parse correctly:

- 'Command line' syntax, ie where a function of string arguments: "save('x','Y')" can be written as "save x Y" in Matlab, but in Prolog, you must use function call syntax with quoted arguments: save('x','Y').
- Matlab's postfix transpose operator "x'" must be written using a different postfix operator "x'" or function call syntax "ctranspose(x)".
- Matlab cell referencing using braces, as in x{1,2} must be written as "cref(x,1,2)".
- Field referencing using dot (.), eg x.thing - currently resolved by using hash (#) operator, eg x#thing.
- Using variables as arrays and indexing them. The problem is that Prolog doesn't let you write a term with a variable as the head functor.

**matlab\_init**(-Key, -Cmd:ml\_expr) [nondet,multifile]  
 Each user-defined clause of **matlab\_init**/2 causes *Cmd* to be executed whenever a new Matlab session is started.

**matlab\_path**(-Key, -Path:list(atom)) [nondet,multifile]  
 Each user-defined clause of **matlab\_path**/2 causes the directories in *Path* to be added to the Matlab path of every new Matlab session. Directories are relative to the root directory as returned by Matlab function **proot**.

**pl2ml\_hook**(+X:term, -Y:ml\_expr) [nondet,multifile]  
 Clauses of **pl2ml\_hook**/2 allow for extensions to the Matlab expression language such that  $V[X] = V[Y]$  if **pl2ml\_hook**(X,Y).

**ml\_open**(+Id:ml\_eng, +Host:atom, +Options:list(\_)) [det]

**ml\_open**(+Id:ml\_eng, +Host:atom) [det]

**ml\_open**(+Id:ml\_eng) [det]

Start a Matlab session on the given host. If *Host*=localhost or the name of the current current host as returned by **hostname**/1, then a Matlab process is started directly. Otherwise, it is started remotely via SSH. *Options* defaults to []. *Host* defaults to localhost.

Start a Matlab session on the specified host using default options. If *Host* is not given, it defaults to localhost. Session will be associated with the given *Id*, which should be an atom. See **ml\_open**/3.

Valid options are

#### **noinit**

If present, do not run initialisation commands specified by **matlab\_path**/2 and **matlab\_init**/2 clauses. Otherwise, do run them.

#### **debug**(In, Out)

if present, Matlab is started in a script which captures standard input and output to files In and Out respectively.

[What if session is already open and attached to *Id*?]

**ml\_close**(+Id:ml\_eng) [det]

Close Matlab session associated with *Id*.

**ml\_exec**(+Id:ml\_eng, +Expr:ml\_expr) [det]

Execute Matlab expression without returning any values.

**ml\_eval**(+Id:ml\_eng, +Expr:ml\_expr, +Types:list(type), -Res:list(ml\_val)) [det]  
 Evaluate Matlab expression binding return values to results list *Res*. This new form uses an explicit output types list, so *Res* can be completely unbound on entry even when multiple values are required.

**ml\_test**(+Id:ml\_eng, +X:ml\_expr(bool)) [semidet]  
 Succeeds if *X* evaluates to true in Matlab session *Id*.

**==**(Y:ml\_vals(A), X:ml\_expr(A)) [det]  
 Evaluate Matlab expression *X* as in **ml\_eval**/4, binding one or more return values to *Y*. If *Y* is unbound or a single ml\_val(\_), only the first return value is bound. If *Y* is a list, multiple return values are processed.

**??**(X:ml\_expr(\_)) [det]  
 Execute Matlab expression *X* as with **ml\_exec**/2, without returning any values.

**???**(X:ml\_expr(bool)) [semidet]  
 Evaluate Matlab boolean expression *X* as with **ml\_test**/2.

**ml\_debug**(+Flag:boolean) [det]  
 Set or reset debug state. **ml\_debug**(true) causes formatted Matlab statements to be printed before being sent to Matlab engine.

**term\_mlstring**(+Id:ml\_eng, +X:ml\_expr, -Y:list(code)) [det]  
 Convert term representing Matlab expression to a list of character codes.

**term\_texatom**(+X:tex\_expr, -Y:atom) [det]  
 Convert term representing TeX expression to a string in atom form.

**wsvar**(+X:ws\_blob(A), -Nm:atom, -Id:ml\_eng) [semidet]  
 True if *X* is a workspace variable in Matlab session *Id*. Unifies *Nm* with the name of the Matlab variable.

**dropmat**(+Id:ml\_id, +Mat:ml\_loc) [det]  
 Deleting MAT file from matbase.

**exportmat**(+Id:ml\_id, +Mat:ml\_loc, +Dir:atom) [det]  
 Export specified MAT file from matbase to given directory.

**matbase\_mat**(+Id:ml\_eng, -X:ml\_loc) [nondet]  
 Listing mat files actually in matbase at given root directory.

**persist\_item**(+X:ml\_expr(A), -Y:ml\_expr(A)) [det]  
 Convert Matlab expression to persistent form not dependent on current Matlab workspace or MX arrays in Prolog memory space. Large values like arrays and structures are saved in the matbase replaced with matbase locators. Scalar values are converted to literal numeric values. Character strings are converted to Prolog atoms. Cell arrays wrapped in the **wsseq**/1 functor are converted to literal form.  
 NB. any side effects are undone on backtracking – in particular, any files created in the matbase are deleted.

**mhhelp**(+Name:atom) [det]  
 Lookup Matlab help on the given name. Equivalent to executing help('X').

**compileoptions**(+Opts:list(ml\_options), -Prefs:ml\_expr(options)) [det]  
 Convert list of option specifiers into a Matlab expression representing options (ie a struct). Each specifier can be a Name:Value pair, a name to be looked up in the **optionset**/2 predicate, a nested list of ml\_options compileoptions :: list(optionset | atom:value | struct) -> struct. NB. option types are as follows:

```

X :: ml_options :- optionset(X,_).
X :: ml_options :- X :: ml_option(_).
X :: ml_options :- X :: list(ml_options).
X :: ml_options :- X :: ml_expr(struct(_)).

ml_option(A) ---> atom:ml_expr(A).

```

**multiplot**(+Type:ml\_plot, +Cmds:list(ml\_expr(-))) [det]

**multiplot**(+Type:ml\_plot, +Cmds:list(ml\_expr(-)), -Axes:list(ml\_val(handle))) [det]

Executes plotting commands in *Cmds* in multiple figures or axes as determined by *Type*.  
Valid types are:

**figs**(Range)

Executes each plot in a separate figure, Range must be P..Q where P and Q are figure numbers.

**vertical**

Executes each plot in a subplot; subplots are arranged vertically top to bottom in the current figure.

**horizontal**

Executes each plot in a subplot; subplots are arranged horizontally left to right in the current figure.

**.**(Type, [link(Axis)])

As for multiplot type *Type*, but link X or Y axis scales as determined by *Axis*, which can be 'x', 'y', or 'xy'.

Three argument form returns a list containing the Matlab handles to axes objects, one for each plot.

**optionset**(+Key:term, -Opts:list(ml\_options)) [semidet,multifile]

Extensible predicate for mapping arbitrary terms to a list of options to be processed by `compileoptions/2`.