The `mpoly` Toolbox

Fritz Keinert
Department of Mathematics
Iowa State University
Ames, IA 50011
keinert@iastate.edu

February 20, 2004

1 Overview

This toolbox defines the data type `mpoly` (matrix Laurent polynomial) and some basic operations on it. It uses object-oriented programming, so it requires MATLAB® version 5 or higher to run. Please report any problems or other suggestions to keinert@iastate.edu.

A matrix Laurent polynomial is a polynomial of the form

$$P(z) = P_{k_0} z^{k_0} + P_{k_0+1} z^{k_0+1} + \cdots + P_{k_1} z^{k_1},$$

where the coefficients P_k are two-dimensional matrices, all of the same size. The exponents are integers, possibly negative. Equivalently, $P(z)$ is a matrix with (Laurent) polynomial entries.

The matrices can be either numerical or symbolic. Implementing symbolic matrices has been more of a headache than anything else; if anything is not working correctly in this toolbox, it will most likely relate to symbolic matrices. Also, symbolic calculations are noticeably slower than numerical calculations, and MATLAB® may need some help in simplifying the result.

This toolbox is part of the multiwavelet toolbox, but it can be installed and used independently.

2 Installation

Install the directories `@mpoly`, `@double` and `@sym` in a directory in the MATLAB® path.

There are several test routines in the `mw` directory. You should run them in the order `test_double`, `test_sym`, `test_mpoly_numerical`, `test_mpoly_symbolic`, in order to verify that everything is working.

Notes:

1. Normally MATLAB® will notice new routines that are added while it is running. However, this does not seem to work for `@`-directories. Remove the installation directory from the path, and add it again.

2. Don't worry if the test routines seem to take a long time to run. They use symbolic computation extensively, which is in fact quite slow, and MATLAB® also needs to compile each routine in the toolbox the first time it is run. Numerical computations, after the initial compilation step, will run much faster.

3 Object-oriented programming in MATLAB®

Up until MATLAB® version 4, there was essentially only one data type: everything was a double-precision, complex, two-dimensional matrix. For example, a string was a $1 \times n$ matrix whose entries were interpreted as ASCII codes; a real scalar was a 1×1 matrix with imaginary part 0.

Since version 5, MATLAB® supports several data types; these include `double` (the default), `char` (text), `sym` (symbolic), `cell` (cell array), `struct` (structure), and others. In addition, the user is allowed to define

new data types and their associated operations. This is called *object-oriented programming*. Data types are called *classes* in MATLAB®.

This feature has two main consequences:

1. Several different subroutines can share the same name. This is called *operator overloading*.

   When you call a subroutine in MATLAB® , for example `sqrt(x)`, MATLAB® looks at the data type of `x`. If the data type is `cell`, for example, it looks through all directories called `@cell` in the path for a routine `sqrt.m`; if that is not found, it looks through all the other directories whose names do not begin with `@`; finally, it looks for a built-in command called `sqrt` (which exists, but is not defined for cell arrays).

   If you want to define a square root operation for cell arrays, for example a routine which applies the standard `sqrt` to each element, you can write one and place it in a directory `@cell` somewhere in the path.

   This implies that routines `@sym/sqrt.m`, `@cell/sqrt.m`, etc. can happily coexist. MATLAB® determines the type of the argument at runtime, and calls the appropriate subroutine.

   Remark: If the argument list includes several different data types for which subroutines of that name have been defined, MATLAB® by default calls the routine belonging to the first nontrivial data type it encounters. You can change precedence with the `inferiorto` and `superiorto` commands.

2. The usual mathematical notation can be used as an alternative to many subroutine calls.

   When you type `x+y`, MATLAB® actually interprets this as a function call `plus(x,y)`. The `plus` function can be overloaded as described above.

   For example, if `x` and `y` are strings, then typing `x+y` causes MATLAB® to look for a routine `@char/plus.m`. You could write such a routine which, for example, concatenates the strings. Other mathematical operators are equivalent to subroutine calls in a similar way. Use of this feature can make your code much more readable.

The standard operators and their equivalent names are given in table 1.

4 The `mpoly` data type

A matrix Laurent polynomial is a polynomial of the form

$$P(z) = P_{k_0} z^{k_0} + P_{k_0+1} z^{k_0+1} + \cdots + P_{k_1} z^{k_1},$$

where the coefficients P_k are two-dimensional matrices, all of the same size. The exponents are integers, possibly negative. Equivalently, $P(z)$ is a matrix with (Laurent) polynomial entries.

Internally, this toolbox represents a matrix polynomial as a structure with the fields

`coef`	A three-dimensional array; `coef(:,:,1)` is P_{k_0}, `coef(:,:,2)` is $P_(k_0+1)$, etc.
`min`	Starting exponent k_0.
`type`	one of `''`, `'symbol'`, `'polyphase'`
`m`	dilation factor
`r`	multiplicity

Remark: The fields `type`, `m`, `r` are used by the multiwavelet package, and are explained there. If you just want to use the `mpoly` package, you can ignore them.

To create an `mpoly` object, you invoke the constructor routine `mpoly(coef,min)`. The coefficient matrices are entered as either a three-dimensional matrix (third dimension = subscript/exponent), or as a cell vector of coefficients.

Example: To create

$$P(z) = P_2 z^2 + P_3 z^3 + P_4 z^4,$$

you can use either

```
P = mpoly(cat(3,P2,P3,P4),2);
```

Table 1: Standard operators and their equivalent function names

Operator notation	Function equivalent
`[A, B, C ...]`	`horzcat(A,B,C...)`
`[A; B; C ...]`	`vertcat(A,B,C...)`
`B=A(i,j,...)`	`subsref(A,S)`
`B=A{i,j,...}`	
`B=A.i`	
`A(i,j,...)=B`	`subsasgn(A,S,B)`
`A{i,j,...}=B`	(type `help substruct` for information on `S`, `B`)
`A.i=B`	
`x(A)`	`subsindex(x,A)`
`A + B`	`plus(A,B)`
`+A`	`uplus(A)`
`A - B`	`minus(A,B)`
`-A`	`uminus(A)`
`A * B`	`mtimes(A,B)`
`A .* B`	`times(A,B)`
`A / B`	`mrdivide(A,B)`
`A ./ B`	`rdivide(A,B)`
`A \ B`	`mldivide(A,B)`
`A .\ B`	`ldivide(A,B)`
`A ^ B`	`mpower(A,B)`
`A .^ B`	`power(A,B)`
`A'`	`ctranspose(A)`
`A.'`	`transpose(A)`
`A & B`	`and(A,B)`
`A | B`	`or(A,B)`
`~A`	`not(A)`
`A == B`	`eq(A,B)`
`A ~= B`	`ne(A,B)`
`A < B`	`lt(A,B)`
`A <= B`	`le(A,B)`
`A > B`	`gt(A,B)`
`A >= B`	`ge(A,B)`
`A`	`display(A)`

or

```
P = mpoly({P2,P3,P4},2);
```

Other examples are

```
P = mpoly(A);
```

which creates the constant polynomial $P(z) = A$, or

```
P = mpoly(1,1);
```

which creates the scalar polynomial $P(z) = z$.

After constructing matrix polynomials, you can use the usual arithmetic operations, which are interpreted in the matrix polynomial sense.

5 Trimming

When you do calculations with matrix polynomials, the result sometimes has unnecessary zero matrices at one or both of the ends. (Zero matrices in the middle have to be stored). The `trim` routine will get rid of leading and trailing zeros.

After using the toolbox myself for a while and experimenting with various setups, I determined that I usually want those extra zeros to go away automatically. Thus, the `trim` routine is invoked automatically at the end of most calculations. You can suppress the trimming by setting the global variable `MPOLY_NOTRIM` to a nonzero value, like this:

```
global MPOLY_NOTRIM
MPOLY_NOTRIM = 1;
```

It is sometimes necessary to introduce extra zero matrices at the ends to achieve a given range of exponents, for example for adding or subtracting matrix polynomials. The `trim` routine can also be used for that. This use of `trim` is not suppressed by `MPOLY_NOTRIM`.

A related question is: What is a zero? Calculations introduce round-off error. If a matrix has entries of order 10^{-15}, is that a zero matrix? My answer is: usually, it is.

There is second global variable `MPOLY_TOLERANCE` which determines what numbers are considered negligible. If this variable is not set, the package uses 10^{-12} as a cutoff. You can set this variable to other values, for example

```
global MPOLY_TOLERANCE
MPOLY_TOLERANCE = 1.e-15;
```

If you set this variable to 0, no rounding to zero occurs. Coefficient matrices which are exactly zero still get trimmed.

Symbolic matrices are not rounded.

6 Subscripting

There are three kinds of subscripting operations in MATLAB®: `P.field`, `P(i)` and `P{i}`. I have implemented all of them for matrix polynomials, with the following effects:

1. `P.field` gives access to the fields actually stored in the data structure, plus some more that are calculated. The following fields are recognized:

stored	`P.min`	smallest exponent
	`P.max`	largest exponent
	`P.coef`	matrix of coefficients
	`P.type`	type (`''`, `'symbol'` or `'polyphase'`)
	`P.m`	dilation factor
	`P.r`	multiplicity
calculated	`P.length`	number of coefficients; this is the same as `max-min+1`
	`P.degree`	Laurent degree; this is the same as `length-1`
	`P.size`	the size of each coefficient matrix

   The syntax `f = get(P,'field')` is equivalent to `f=P.field`. The syntax `set(P,'field',f)` is equivalent to `P.field=f`.

   Only the stored fields can be used on the left-hand side of an assignment statement.

2. `P(x)` evaluates the matrix polynomial at the point(s) `x`. The argument can be a scalar, a matrix, or a matrix polynomial. The effect is identical to `mpolyval(P,x)`.

3. `P(i,j)` is used to select a submatrix. For example, if

$$P(z) = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} + \begin{pmatrix} 3 & 2 & 1 \\ 6 & 5 & 4 \\ 9 & 8 & 7 \end{pmatrix} z + \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} z^2,$$

   then

$$\texttt{P(1:2,2:3)} = \begin{pmatrix} 2 & 3 \\ 5 & 6 \end{pmatrix} + \begin{pmatrix} 2 & 1 \\ 5 & 4 \end{pmatrix} z + \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} z^2.$$

4. `P{n}` is used to select the coefficient matrix that goes with z^n. For example, for the same `P` as above,

$$\texttt{P\{1\}} = \begin{pmatrix} 3 & 2 & 1 \\ 6 & 5 & 4 \\ 9 & 8 & 7 \end{pmatrix}.$$

   If `n` is outside the range `P.min` to `P.max`, a zero matrix of appropriate size is returned.

   It is not possible to use more than a single subscript in `{}` subscripting. To select a subpolynomial with a contiguous range of exponents, use the `trim` routine.

Remark: Note the two different kinds of `()` subscripting: if there is a single subscript (which can be a matrix), it does point evaluation. If there are two subscripts, it selects a submatrix. So, `P(1,2)` (two subscripts; selects scalar sub-polynomial) has a completely different effect from `P([1,2])` (one subscript; evaluates `P` at two points).

7 List of Routines

A list of routines is given in tables 2 and 3. For details about the routines, use the `help` function inside MATLAB®. You can also look at the test routines and routine `example_mpoly` for examples.

Table 2: List of Routines (Part 1)

Constructor/Conversion Routines	`mpoly`	constructor routine
	`polyphase`	conversion to multiwavelet polyphase matrix
	`symbol`	conversion to multiwavelet symbol
Arithmetic Operations	`ctranspose`	complex conjugate transpose (replaces z by $1/z$)
	`inv`	inverse
	`ldivide`	elementwise left division
	`minus`	subtraction
	`mldivide`	matrix left division
	`mpower`	matrix power
	`mrdivide`	matrix right division
	`mtimes`	matrix product
	`plus`	addition
	`power`	elementwise power
	`rdivide`	elementwise right division
	`times`	elementwise product
	`transpose`	real transpose
	`uminus`	unary minus
	`uplus`	unary plus

Table 3: List of Routines (Part 2)

Comparison Operators	`eq`	equality
	`isconstant`	checks whether P is a constant matrix
	`isidentity`	checks whether P is an identity matrix
	`ismonomial`	checks whether P is a monomial
	`isnumeric`	checks whether P is numeric (see note 1.)
	`issymbolic`	checks whether P contains symbolic variables (see note 1.)
	`iszero`	checks whether P is a zero matrix
	`ne`	non-equality
Access To Properties	`get`	`x=get(P,field)` is the same as `x=P.field`
	`length`	`length(P)` is the same as `P.length`
	`set`	`set(P,field,x)` is the same as `P.field=x`
	`size`	`size(P)` is the same as `P.size`
Subscripting/Concatenation	`end`	for use in subscripts, as in `P(:,3:end)`
	`horzcat`	horizontal concatenation
	`subsasgn`	subscripting on left-hand side of assignment
	`subsindex`	using an `mpoly` as a subscript
	`subsref`	subscripting on right-hand side of assignment
	`vertcat`	vertical concatenation
Standard Routines Extended To Mpoly	`ceil`	round towards ∞
	`diag`	set off-diagonal terms to zero (see note 2.)
	`double`	convert symbolic to numeric
	`expand`	expand symbolic expressions
	`fix`	round towards 0
	`floor`	round towards $-\infty$
	`kron`	Kronecker product
	`reshape`	reshape coefficient matrices
	`round`	round towards nearest integer
	`simplify`	try to simplify symbolic expressions
	`sym`	convert numeric to symbolic
	`tril`	set terms in upper triangle to zero
	`triu`	set terms in lower triangle to zero
Other	`det`	determinant
	`display`	display routine
	`longinv`	long inversion (based on Kramer's rule)
	`match_type`	propagate type of operands (`'symbol'` or `'polyphase'`) to the result of a computation
	`moment`	compute moments
	`mpolyval`	evaluate `mpoly` at given points
	`reverse`	replace z by $1/z$
	`trim`	trim zero matrices at ends

Notes:

1. Routine `issymbolic` is not the opposite of `isnumeric`. I distinguish three kinds of allowed entries: numbers, symbolic constants (symbolic expressions which have a numerical value), and symbolic variables (symbolic expressions which contain variable names). Some operations (such as determining whether a matrix is singular) are possible for symbolic constants, but not for symbolic variables in general. Here is a reference table:

`x`	`isnumeric(x)`	`issymbolic(x)`
`sqrt(2)`	true	false
`sym('sqrt(2)')`	false	false
`2+a`	false	true

2. The `diag` routine is similar to the built-in `diag`, but not quite the same. The statement `D=diag(P)` returns a vector `D` if `P` is a matrix, but it returns a diagonal matrix polynomial `D` if `P` is of type `mpoly`. The statement `P=diag(D)` produces a diagonal matrix `P` if `D` is a vector; if `D` is a vector polynomial, `P` would be its diagonal, which is a scalar polynomial (the first entry).

   However, you can use `diag(D,k)` to extract the kth diagonal (again, as a matrix), and likewise for `tril`, `triu`.

8 More Than You Wanted To Know About Subscripting

This section explains some aspects of subscripting in MATLAB® in more detail, including some frustrating experiences I had in implementing subscripts. This section is only useful if you want to read and understand the source code. It is not necessary to understand this section in order to apply the toolbox.

Assume `P` is of class `mpoly`. A subscripting operation of the form `P.field`, `P(subscript)`, `P{subscript}` on the right-hand side of an assignment statement invokes routine `subsref`; subscripting on the left-hand side of an assignment statement invokes `subsasgn`. If `P` itself is used as a subscript, as in `A(P)`, it invokes `subsindex`.

Normally, if you do subscripting on an `mpoly` object, MATLAB® will look for the subscripting routines in directory `@mpoly`. From routines inside the `@mpoly` subdirectory, however, MATLAB® always calls the built-in subscripting routines, not the user-written routines. That makes sense, since you could easily get into an infinite loop if you don't watch out.

However, this behavior was different in early versions of MATLAB® 5. Also, it changed from version to version with no documentation. Experimentally, I have determined that a subscripting operation from a routine inside `@mpoly` calls the following `subsref` routine:

subscripting operation	MATLAB® version 5.1	5.2	5.3 and higher
`.`	built-in	built-in	built-in
`()`	user	built-in	built-in
`{}`	user	user	built-in

I think I have written the code in such a way that it works in all flavors of MATLAB® now.