User-Manual of the MacaulayLab Toolbox

Christof Vermeersch

May 21, 2025

Abstract This document contains the user-manual of the MacaulayLab toolbox (version `1.0`). Via several examples, the user is guided through the available functions of the software. The user-manual explains how to solve multivariate polynomial systems and rectangular multiparameter eigenvalue problems.

Quick start For those users that are not interested in the full user-manual of MacaulayLab, but just want to solve problems as soon as possible, I recommend jumping to section 1.3 for a minimal introduction to the software. This part of the text should provide you with all the necessary information (and nothing more) to solve your problem.

Contents

1 Getting started

MacaulayLab is a Matlab toolbox that features algorithms to solve multivariate polynomial systems and rectangular multiparameter eigenvalue problems. Before using MacaulayLab, you need to download the zip archive of the toolbox from the website `www.macaulaylab.net` and unzip MacaulayLab to any directory. You could also clone the latest version of the repository on Github. The repository is intended for developers and collaborators. It also contains additional test files, beta versions of the toolbox, and previous releases. This user-manual is based on version `1.0` of the software. You can check the current version of MacaulayLab that you use via `information("version")`.

1.1 Installation

Afterward unzipping MacaulayLab to a directory (or cloning the repository), you can browse to that location in Matlab and add the path.

Code 1. It is easy to add MacaulayLab to your path:

```
>> addpath(genpath(pwd));
>> savepath;
```

1.2 Help and documentation

The different functions of MacaulayLab are well-documented. Using the function `help function` in the command line displays more information about the functionality and interface of that function.

Code 2. The documentation of the `nbmonomials` function:

```
>> help nbmonomials
nbmonomials     Number of monomials.
    s = nbmonomials(d,n) returns the number of monomials in
    the monomial basis for n variables and maximum total
    degree d.

    s = nbmonomials(...,blocksize) takes into account the
    blocksize.

    See also monomials.
```

1.3 Quick start

The easiest way to represent a system of multivariate polynomials is by considering a matrix for every polynomial of the system, where the first column corresponds to the coefficients of the polynomials and the remaining columns represent the powers of the variables in the corresponding monomials. These matrices are combined in a cell array and given to the `systemstruct` constructor.

Code 3. We start by constructing a system of two bivariate polynomial equations.

```
>> p1 = [2 2 0; -3 0 1; 1 0 0];
>> p2 = [1 2 0; 1 0 2; 16 0 0];
>> eqs = {p1, p2};
>> problem = systemstruct(eqs);
```

Similarly, a rectangular multiparameter eigenvalue problem can be represented by a cell array that contains all its coefficient matrices and a matrix that describes the monomial that belongs to each coefficient matrix. This cell array and matrix are given to the `mepstruct` constructor.

Code 4. Similarly, we can construct a linear two-parameter eigenvalue problem. Contrary to `systemstruct`, `mepstruct` requires information about the support.

```
>> A00 = randn(4,3); A10 = randn(4,3); A01 = randn(4,3);
>> mat = {A00, A10, A01};
>> supp = [0 0; 1 0; 0 1];
>> problem = mepstruct(mat,supp);
```

Solving the problem requires only one additional line of code.

Code 5. Given the problem, it is possible to obtain its solutions without any additional information. The solutions are given in a `solutionstruct`.

```
>> solutions = macaulaylab(problem);
```

The solutions are given in a solution stucture. If you want to use the numerical values of the solutions, you can access them easily via `num(solutions)`.

Code 6. You access the numerical values inside a `solutionstruct` via `num`:

```
>> values = num(solutions)

values =

    2.2876 + 0.0000i    14.0549 + 0.0000i
    2.7729 + 4.1445i     1.5091 - 1.9992i
    2.7729 - 4.1445i     1.5091 + 1.9992i
    1.0350 - 0.8553i    -3.1730 - 1.5772i
    1.0350 + 0.8553i    -3.1730 + 1.5772i
    0.5146 + 0.0000i     0.2825 + 0.0000i
```

1.4 Tests to check all functionality

The repository of MacaulayLab contains a large set of unit tests to check the different functions. This allows the user to change the code and experiment with the different functions, but at all times the user can check whether everything still works. You can find the test suite in the folder `Test` (only for those people who downloaded the development version of the toolbox). In order to run all the tests, simply use Matlab's function `runtests`.

Code 7. After moving to the correct folder, you can run all the tests:

```
>> cd Test/
>> results = runtests(pwd,"IncludeSubfolders",true);
```

2 Representation of a problem

In order to keep the toolbox user-friendly, describing a problem is kept very simple. MacaulayLab revolves around two different types of problems: multivariate polynomial systems and rectangular multiparameter eigenvalue problems. Both problem types are internally represented by the same class `problemstruct`: all necessary information is stored in the cell arrays `coef` and `supp`, where each cell of `coef` and `supp` contains the coefficients/coefficient matrices and support of one polynomial (matrix) equation, respectively. Although it is possible to submit the problem directly in its internal representation, the sub-classes `systemstruct` and `mepstruct` provide constructors to set-up the specific problems more easily (Figure 1).

This toolbox works perfectly with the database of test problems that can be found at https://github.com/christofvermeersch/database. More infor-

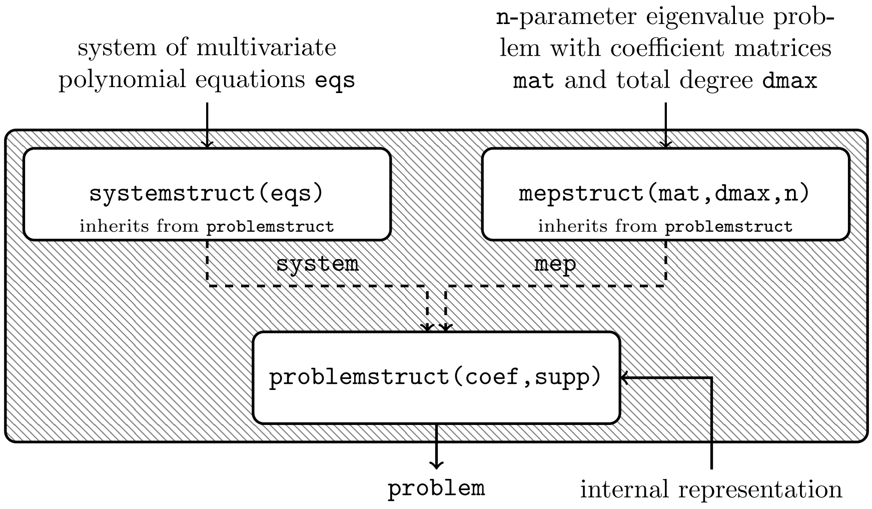
system of multivariate polynomial equations eqs
n-parameter eigenvalue problem with coefficient matrices mat and total degree dmax
systemstruct(eqs)
inherits from problemstruct
mepstruct(mat,dmax,n)
inherits from problemstruct
system
mep
problemstruct(coef,supp)
problem
internal representation

Figure 1: Representation of a system of multivariate polynomial equations or rectangular multiparameter eigenvalue problem in MacaulayLab. Both problem types are internally represented by the same `problemstruct`: all necessary information is stored in the cells `coef` and `supp`. The sub-classes `systemstruct` and `mepstruct` provide constructors to set-up the problems more easily, but it is also possible to submit the problem directly in the internal representation.

mation about a specific multivariate polynomial system or rectangular multiparameter eigenvalue problem can be obtained via `help problem` or `disp(problem)`. Most problems in the database already contain some information about the number of affine solutions, total number of solutions, required time to solve the problem, etc.

2.1 Multivariate polynomial systems

The natural way of representing a single polynomial is via a row vector with its coefficients. The coefficients in that row vector are ordered according to a particular monomial ordering. For example, the polynomial $p(x, y, z) = x^2 + 2yz + 3$ as a row vector corresponds to

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 3 \end{bmatrix}, \tag{1}$$

in the GREVLEX ordering, where the leading term is the first element of the row vector. Of course, this representation contains many zeros when the polynomial is sparse, especially for high degrees and many variables. A more efficient approach to represent a polynomial is by considering a matrix, where the first column corresponds to the coefficients of the polynomial and the remaining columns represent the powers of the variables in the corresponding monomials, i.e.,

$$p_i\left(\boldsymbol{x}\right) \leftrightarrow \begin{bmatrix} c_1 & \alpha_{11} & \cdots & \alpha_{1n} \\ c_2 & \alpha_{21} & \cdots & \alpha_{2n} \\ \vdots & \vdots & & \vdots \\ c_N & \alpha_{N1} & \cdots & \alpha_{Nn} \end{bmatrix}, \tag{2}$$

with c_i the coefficients of the polynomial and α_{ij} the power of the variable x_j for that ith coefficient. The polynomial $p(x, y, z) = x^2 + 2yz + 3$ is represented by a matrix with three rows and four columns (three variables and the coefficients):

$$\begin{bmatrix} 1 & 2 & 0 & 0 \\ 2 & 0 & 1 & 1 \\ 3 & 0 & 0 & 0 \end{bmatrix}. \tag{3}$$

When the monomial ordering is set, it is possible to switch between these two representations

Code 8. A polynomial in its matrix representation can be expanded into a row vector via `expansion(poly)`

```
>> P = [1 2 0 0; 2 0 1 1; 3 0 0 0];
>> p = expansion(P)

p =
     1     0     0     0     2     0     0     0     0     3
```

The function `contraction(p,d,n)` goes in the other direction:

```
>> contraction(p,2,3)

ans =
     1     2     0     0
     2     0     1     1
     3     0     0     0
```

A system of (multivariate) polynomials is represented internally by two cell arrays `coef` and `supp`. However, you do not need to worry about this internal representation. Via a cell array and the `systemstruct` constructor, multiple polynomials can be combined into that problem representation.

Code 9. By combining different polynomials in `systemstruct`, a system can be constructed in MacaulayLab.

```
>> p1 = [2 2 0 0; -3 0 1 3; 1 0 0 0];
>> p2 = [1 2 0 0; 1 0 2 0; 1 0 0 2; 16 0 0 0];
>> p3 = [1 1 1 1; 2 0 0 0];
>> eqs = {p1, p2, p3};
>> system = systemstruct(eqs);
```

The `systemstruct` constructor can also work with symbolic polynomials.

Code 10. When a multivariate polynomial system is defined via symbolic polynomials, the `systemstruct` can still be used to create a problem in MacaulayLab.

```
>> syms x y;
>> p1 = 2*x^2 + 3*x*y;
>> p2 = x^2*y^3 - 1;
>> symboliceqs = {p1, p2};
>> system = systemstruct(symboliceqs);
```

Suppose that you want to keep the coefficients and support in two separate cell arrays, then you could use a column vector with the coefficients and a matrix with the support per polynomial. You could avoid using the `systemstruct` constructor and submit the system directly using the internal representation of the toolbox. The cell array `coef` contains per equation a cell with a column vector of the coefficients, while each cell of `supp` has a matrix with the corresponding support for these coefficients.

Code 11. A system can also be constructed directly by giving the internal representation to `problemstruct`.

```
>> coef1 = [2; -3; 1];
>> supp1 = [2 0 0; 0 1 3; 0 0 0];
>> coef2 = [1; 1; 1; 16];
>> supp2 = [2 0 0; 0 2 0; 0 0 2; 0 0 0];
>> coef3 = [1; 2];
>> supp3 = [1 1 1; 0 0 0];
>> coef = {coef1, coef2, coef3};
>> supp = {supp1, supp2, supp3};
>> system = problemstruct(coef,supp);
```

In order to create a random dense system, `randsystem(s,dmax,n)` can be used, which generates a system of `s` multivariate polynomials in `n` variables that has random coefficients for every monomial up to total degree `dmax`. Via additional arguments, it is possible to choose a specific support or complex coefficients.

2.2 Rectangular multiparameter eigenvalue problems

Similarly, a rectangular multiparameter eigenvalue problem can be represented by a cell array that contains all the coefficient matrices and a matrix that describes the monomial that belongs to each coefficient matrix. For example, the linear two-parameter eigenvalue problem $\boldsymbol{\mathcal{M}}(\boldsymbol{\lambda})\,\boldsymbol{z} = (A_{00} + A_{10}\lambda_1 + A_{01}\lambda_2)\,\boldsymbol{z} = \mathbf{0}$ has three coefficient matrices. This cell array and the support are then given to the `mepstruct` constructor.

Code 12. Contrary to `systemstruct`, `mepstruct` requires information about the support of the coefficient matrices.

```
>> A00 = randn(4,3); A10 = randn(4,3); A01 = randn(4,3);
>> mat = {A00, A10, A01};
>> supp = [0 0; 1 0; 0 1];
>> mep = mepstruct(mat,supp);
```

Again, you could decide to enter the problem directly in its internal representation. A single rectangular multiparameter eigenvalue problem consists of one cell in `coef` and one cell in `supp`. The cell in `coef` is a three-dimensional array where the coefficient matrices are stacked along the first dimension, while the cell in `supp` contains a two-dimensional array with the support (each row corresponds to the monomials that belong to that coefficient matrix).

Code 13. A multiparameter eigenvalue problem can also be constructed directly by giving the internal representation to `problemstruct`.

```
>> coef = {tensorization(mat)};
>> supp = {[0 0; 1 0; 0 1]};
>> multiparameter eigenvalue problem =
problemstruct(coef,supp);
```

The function `tensorization` put the coefficient matrix into the correct tensor format.

Quickly constructing a rectangular multiparameter eigenvalue problem with random coefficient matrices is also very easy with `randmep(dmax,n,k,l)`, where `dmax` is the maximum total degree, `n` is the number of eigenvalues, `k` is the number of rows, and `l` is the number of columns of the problem. This function also allows arguments to choose a support or complex values.

2.3 Information about a problem

After defining a system, it is possible to retrieve information about a problem via the overloaded `disp` or the more elaborated `info`. More specific information can be retrieven via `properties` or directly via dot indexing, while `structure` displays the structure of the problem.

Code 14. You can get more information about a problem via `disp`:

```
>> disp(problem)

    2-parameter eigenvalue problem with total degree 4
```

The specific information can be accessed via `properties` or dot indexing:

```
>> [s,dmax,n,di,nnze,k,l] = properties(problem);
>> s = problem.s;
```

If you want to have a quick view of the structure, then you can use `structure` or the more elaborated `info`.

```
>> info(problem)

    2-parameter eigenvalue problem with total degree 4

        (A00 + A10 λ1 + A13 λ1 λ2^3 ) z = 0

    coefficient matrices are 4 x 3 matrices
    sparsity factor is 0.20
```

3 Solutions via the (block) Macaulay matrix

MacaulayLab uses a similar approach to solve both problem types. Many of the functions are, therefore, re-used when solving the different problems. We give a step-by-step solution approach (section 3.1), but the user is more likely to use the direct solution approach (section 3.2).

3.1 Step-by-step solution approach

Building the (block) Macaulay matrix generated by the problem is probably the first step you take after defining the problem. Since both problems are represented internally by the same data structure, the same function can be used to build the Macaulay matrix for a multivariate polynomial system or block Macaulay matrix for a rectangular multiparameter eigenvalue probelm. The function `macaulay(problem,d)` builds the (block) Macaulay matrix of degree `d` that incorporates the `problem`.

Code 15. A Macaulay matrix of degree d can easily be constructed via `macaulay(system,d)`.

```
>> M = macaulay(redeco6,10);
```

With the same function, a block Macaulay matrix of degree d can be constructed: `macaulay(mep,d)`.

```
>> N = macaulay(hkp2,5);
```

The default (block) Macaulay matrix solution approach in this toolbox uses a basis matrix of the right null space the (block) Macaulay matrix. This matrix can be computed directly via the standard approach `null`, via the recursive approach `macaulayupdate` and `nullrecrmacualay`, or via the sparse approach `nullsparsemacaulay`.

Code 16. The standard approach to compute a basis matrix of the right null space is via `null(Z)`:

```
>> M = macaulay(system,5);
>> Z = null(M);
```

Alternatively, the basis matrix can also be built recursively via `macaulayupdate(M,problem,d)` and `nullrecrmacaulay(Z,Y)`, where `Y` is the difference between the two (block) Macaulay matrices:

```
>> M = macaulay(system,2);
>> Z = null(M);
>> for d = 3:5
        rows = size(M,1);
        M = macaulayupdate(M,system,d);
        Z = nullrecrmacaulay(Z,M(rows+1:end,:));
   end
```

It is also possible to use `nullsparsemacaulay(Z,problem,d)`, which avoids the construction of the (block) Macaulay matrix:

```
>> M = macaulay(system,2);
>> Z = null(M);
>> for d = 3:5
       Z = nullsparsemacaulay(Z,system,d);
   end
```

Instead of performing these iterations by hand, you can call `nullitermacaulay` directly.

Once you have a basis matrix of the right null space, you want to look for the standard monomials and determine the gap zone. Of course, you can also determine the gap directly via `gap`.

Code 17. To determine the standard monomials and the degree of the gap zone:

```
>> c = stdmonomials(Z);
>> [dgap, ma] = gapstdmonomials(c,d,n);
```

Or, directly, via `gap` (which also gives the number of affine solutions):

```
>> [dgap, ma] = gap(Z,n);
```

With this information, the column compression to remove the solutions at infinity is quite straightforward:

Code 18. You can perform a column compression via:

```
>> W11 = columncompr(Z,dgap,n);
```

Solving the problem can be done via performing shifts in the right null space and finding the eigenvalues of (generalized) multiplication maps. `multmapnull` constructs the (generalized) multiplication maps from the right null space that yield the solutions of the problem.

Code 19. The function `multmapnull(Z,K,G,idx)` considers the shift problems defined in `G` and uses these shifts in the right null space `Z` to set up the (generalized) multiplication maps. `idx` contains the indices of the rows that are shifted. For example,

```
>> [A,B] = multmapnull(Z,K,[2 1 2],[1 2 3]);
```

returns two matrices `A` and `B` that are constructed by shifting the first three rows inside the right null space `Z` by the polynomial $2x_1x_2^2$. The matrix `K` describes with which monomials the rows of `Z` correspond. Consequently,

```
>> values = eig(A,B);
```

gives the evaluations of this polynomial in the solutions contained in the right null space.

3.2 Direct solution approach

Of course, there is a solver implemented that takes care of all these intermediate steps and checks whether the right null space of the (block) Macaulay matrix can accommodate the shift polynomial. Furthermore, it is also possible to consider the column space instead of the right null space of the (block) Macaulay matrix. You can use `macaulaylab(problem)` to solve a problem via the default approach (which is currently a block-wise sparse null space based approach).

Code 20. Two examples of using `macaulaylabproblem`:

```
>> commonroots = macaulaylab(redeco6);
>> eigenvalues = macaulaylab(hkp2);
```

To avoid unpleasant surprises, it is recommended to set a maximum degree for the (block) Macaulay matrix. By default, the maximum degree is set to 100.

Code 21. By using `macaulaylab(problem,dend)`, the maximum degree of the (block) Macaulay matrix is set to `dend = 20`.

```
>> roots = macaulaylab(redeco6,20);
```

The solver has many options and outputs a lot of information. table 1 contains an overview of the different options, but the default options should result in an acceptable trade-off between computation speed and numerical stability. The options can be set via supplementing key-value pairs. For example, asking the solver for a iterative enlargement of the (block) Macaulay matrix can be achieved by using the argument `enlarge = "iterative"`.

Code 22. If you want to combine the key-value pairs with a maximum degree for the (block) Macaulay matrix, then `dend` has to be given before the key-value pairs.

```
>> dend = 20;
>> solutions =
macaulaylab(problem,dend,enlarge="iterative");
```

The solution and output details are given in two structures, namely a `solutionstruct` and `outputstruct`, respectively. If you want to use the numerical values of the solutions, you can access them easily via `num(solutions)`.

Code 23. You access the numerical values inside a `solutionstruct` via `num`:

```
>> values = num(solutions)

values =

    2.2876 + 0.0000i    14.0549 + 0.0000i
    2.7729 + 4.1445i     1.5091 - 1.9992i
    2.7729 - 4.1445i     1.5091 + 1.9992i
    1.0350 - 0.8553i    -3.1730 - 1.5772i
    1.0350 + 0.8553i    -3.1730 + 1.5772i
    0.5146 + 0.0000i     0.2825 + 0.0000i
```

4 Monomial order and polynomial basis

MacaulayLab is implemented independently from the monomial order and polynomial basis, which means that you can supply a certain monomial order or polynomial basis and use all the functions without any adaptations. Choosing a certain monomial order is done by giving a function that determines the position of a monomial to the solver, e.g., `grevlex` and `grnlex`. The definition of a basis requires the user to supply a function that implements a definition of the basis, which consists of describing how the product of two monomials in that basis look like and gives the parameters of the three-term recursion definition. These functions are given to MacaulayLab as function handles.

Code 24. It is possible to construct the Macaulay matrix in any polynomial basis or monomial order. `basis` and `order` should be two functions that implement the basis multiplication and the position of a monomial in the monomial ordering.

Table 1: Overview of the different options that can be used by the MacaulayLab solver. Section 4 gives more information about the options to set the monomial ordering and polynomial basis. For the full list of available options and possible inputs, which change when other techniques become available, we recommend to consult the documentation of `macaulaylab`. (The variables c_i are random coefficients.)

option	**type**	**default**
`tol`	`double`	1×10^{-10}
`clustertol`	`double`	1×10^{-10}
`polynomial`	`double`	$\sum_{i=1}^{n} c_i x_i$
`basis`	`function_handle`	`@monomial`
`order`	`function_handle`	`@grevlex`
`subspace`	`string`	`"null space"`
`blocks`	`string`	`"block-wise"`
`enlarge`	`string`	`"sparse"`
`check`	`string`	`"iterative"`
`clustering`	`logical`	`true`
`posdim`	`logical`	`false`
`verbose`	`logical`	`false`

```
>> basis = @chebyshev; % Chebyshev monomial basis
>> order = @grnlex; % grnlex monomial order
>> M = macaulay(problem,d,basis,order);
```

The toolbox has some pre-implemented functions for the monomial order and polynomial basis:

- Monomial order: `grevlex`, `grnlex`, `grlex`, and `grvlex`.
- Polynomial basis: `monomial`, `chebyshev`, and `legendre`.

5 Overview of all functions in the toolbox

When working with multivariate polynomial systems and rectangular multiparameter eigenvalue problems, some other functions might come in handy. The list below gives an overview of all functions included in the toolbox.

Toolbox

- `macaulaylab` - Numerical solver that uses the Macaulay matrix.
- `gettingstarted` - is a live script.

Toolbox/Basis

- `chebyshev` - Multivariate product Chebyshev basis.
- `legendre` - Multivariate product Legendre basis.
- `monomial` - Standard multivariate monomial basis.

Toolbox/Order

- `grevlex` - Graded reverse lexicographic monomial order.
- `grlex` - Graded lexicographic monomial order.
- `grnlex` - Graded negative lexicographic monomial order.
- `grvlex` - Graded inverse lexicographic monomial order.

Toolbox/Structures

- `problemstruct` - Class for generic problems.
- `mepstruct` - Class for multiparameter eigenvalue problems.
- `systemstruct` - Class for multivariate polynomial systems.
- `outputstruct` - Class for gathering all the output details.
- `solutionstruct` - Class for gathering all the solution data.

Toolbox/Internal
Different implemented solution bounds

- `bezout` - Bézout number for a multivariate polynomial system.
- `bkk` - Solution bound for a multivariate polynomial system.
- `hkp` - Solution bound for a multiparameter eigenvalue problem.
- `kushnirenko` - Solution bound for a multivariate polynomial system.
- `shapiro` - Solution bound for a multiparameter eigenvalue problem.

Function to check, manipulate, and generate problems

- `sparsity` - Sparsity factor of a problem.
- `contraction` - Matrix representation of a polynomial.
- `conversion` - Multiparameter eigenvalue problem conversion.
- `expansion` - Row vector representation of a polynomial.
- `tensorization` - Conversion from matrix to tensor representation.

- `randmep` - Dense multiparameter eigenvalue problem.
- `randsystem` - Dense system with random coefficients.

Everything related to constructing special matrices

- `companion` - Construction of the Companion matrix.
- `vandermonde` - Multivariate (block) Vandermonde matrix.
- `rowrecomb` - Row combination matrix.
- `rowshift` - Matrix of shifts.
- `macaulay` - Construction of the (block) Macaulay matrix.
- `macaulayupdate` - Update of the (block) Macaulay matrix.
- `nullitermacaulay` - Iterative null space basis matrix computation.
- `nullrecrmacaulay` - Recursive null space update of a Macaulay matrix.
- `nullrecrrow` - Recursive null space update of a (block) row matrix.
- `nullsparsemacaulay` - Sparse null space update of a Macaulay matrix.

Functions necessary to solve the problems

- `columncompr` - Column compression.
- `gap` - Gap zone of the basis matrix.
- `gapstdmonomials` - Gap zone via the standard monomials.
- `multmapcolumn` - Generalized multiplication matrices (column space).
- `multmapnull` - Generalized multiplication matrices (right null space).
- `multiplicity` - Clustered multiple solutions.
- `value` - Evaluation of the (matrix) equation(s) in a certain point.
- `residuals` - Solution residuals for a problem.

Other functions

- `shift` - Result of shifting two monomials.
- `position` - Position of monomial.
- `monomials` - Matrix with all the monomial vectors.
- `stdmonomials` - Standard monomials.
- `nbmonomials` - Number of monomials.