Chapter 1

User-Manual of the MacaulayLab Toolbox

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

Outline. The user-manual helps the user getting started in Section 1.1. Next, Section 1.2 explains how to represent a problem, being a system of multivariate polynomial equations or a rectangular multiparameter eigenvalue problems. Afterwards, Sections 1.3 and 1.4 show how to solve the problems via the (block) Macaulay matrix and how to change the monomial ordering or polynomial basis, respectively. Finally, some other useful functions of the toolbox are highlighted in Section 1.5

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, we recommend jumping to Section 1.1.3 for a minimal introduction to the software. This section should provide you with all the necessary information (and nothing more) to solve your problem.

1.1 Getting started

MacaulayLab is a Matlab toolbox that features algorithms to solve systems of multivariate polynomial equations and rectangular multiparameter eigenvalue problems (MEPs). It also contains a database with many test 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 stable repository. This user-manual is based on version `1.0` of the software. You can check the current version of MacaulayLab that you use via `ver MacaulayLab`.

1.1.1 Installation

Afterwards, you can browse to that location in Matlab and add the path (via the function `genpath` you also immediately add the subdirectories to the path).

Code 1.1. It is easy to add MacaulayLab to your path:

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

1.1.2 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 1.2. The documentation of the `nbmonomials` function:

```
>> help nbmonomials

nbmonomials calculates the number of monomials.

    [s] = nbmonomials(d,n) calculates the number of
    monomials s in the monomial basis for n variables and
    maximum total degree d.

Input/output variables:

    s: [int] number of monomials in the monomial basis.
    d: [int] maximum total degree of the monomials.
    n: [int] number of variables of the monomials.

See also monomialsmatrix.
```

1.1.3 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 1.3. 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 MEP can be represented by a cell array that contains all the coefficient matrices in the correct monomial ordering (including zero matrices). This cell array, together with the total degree and number of variables of the problem, is then given to the `mepstruct` constructor.

Code 1.4. Similarly, we can construct a linear two-parameter eigenvalue problem. Contrary to `systemstruct`, `mepstruct` requires information about the maximum total degree `dmax` and number of eigenvalue `n`.

```
>> dmax = 1; n = 2;
>> A00 = randn(4,3); A10 = randn(4,3); A01 = randn(4,3);
>> mat = {A00, A10, A01};
>> problem = mepstruct(mat,dmax,n);
```

Solving the problem requires only one additional line of code.

Code 1.5. Given the problem, it is possible to obtain its solutions without any additional information.

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

1.1.4 Tests to check all functionality

MacaulayLab contains a set of tests in order to check if all the Matlab functions behave as expected. This allows the user to change and experiment with the different functions, but at all times the user can check if everything still works. You can find the test suite in the folder `Tests`. In order to run all the tests, simply use Matlab's function `runtests`.

Code 1.6. After moving to the correct folder, you can run all the tests:

```
>> cd Tests/
>> results = runtests(`outputdetail',0);
```

1.2 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: systems of multivariate polynomial equations and rectangular MEPs. 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 also 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 1.1).

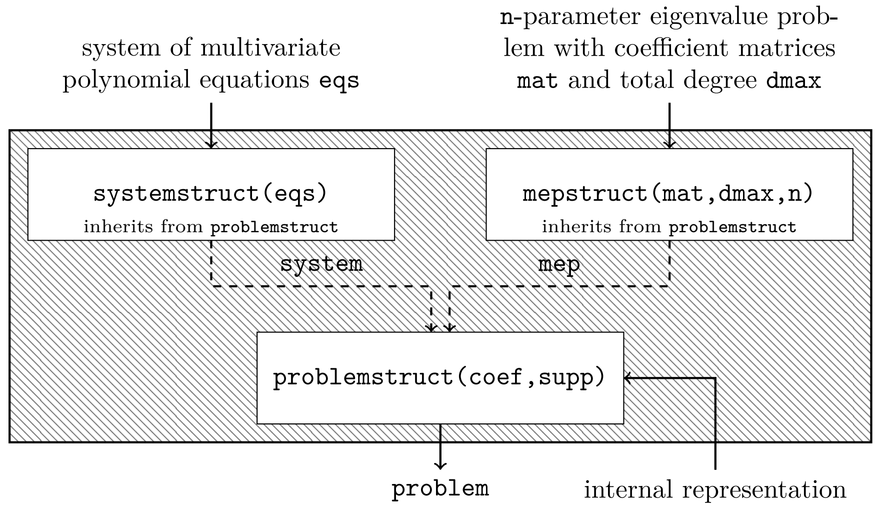
system of multivariate polynomial equations eqs
n-parameter eigenvalue problem with coefficient matrices mat and total degree dmax
systemstruct(eqs)
inherits from problemstruct
mepstruct(mat,dmax,n)
inherits from problemstruct
system
mep
problemstruct(coef,supp)
problem
internal representation

Figure 1.1. Representation of a system of multivariate polynomial equations or rectangular MEP 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.

1.2.1 Systems of multivariate polynomial equations

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 $p(x,y,z) = 1x^2 + 2yz + 3$ as a row vector corresponds to (in the graded inverse lexicographic (GRINVLEX) ordering)

$$\begin{bmatrix} 3 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 2 & 0 \end{bmatrix}. \tag{1.1}$$

Of course, this representation contains many zeros when the polynomial is sparse, especially for high degrees and number of variables. A more efficient approach to represent a polynomial is by considering a matrix, 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, i.e.,

$$p_i(\boldsymbol{x}) \leftrightarrow \begin{bmatrix} c_1 & \alpha_{11} & \cdots & \alpha_{1n} \\ c_2 & \alpha_{21} & \cdots & \alpha_{2n} \\ \vdots & \vdots & & \vdots \\ c_N & \alpha_{N1} & \cdots & \alpha_{Nn} \end{bmatrix}, \tag{1.2}$$

with c_i the coefficients of the polynomial and αij the power of the variable x_j for that ith coefficient. The polynomial $p(x,y,z) = 1x^2+2yz+3$ is represented by a matrix with three rows and four columns (three variables and the coefficients):

$$\begin{bmatrix} 1 & 2 & 0 & 0 \\ 2 & 0 & 1 & 1 \\ 3 & 0 & 0 & 0 \end{bmatrix}. \tag{1.3}$$

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

Code 1.7. A polynomial in its matrix representation can be expanded into a row vector via `expandedpoly(poly)`

```
>> p = [1 2 0 0; 2 0 1 1; 3 0 0 0]

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

>> pexpanded = expandedpoly(p)

pexpanded =
     3     0     0     0     1     0     0     0     2     0
```

The function `contractedpoly(poly,d,n)` goes in the other direction :

```
>> contractedpoly(pexpanded,3,2)

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 1.8. 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);
```

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 `coef` contains per equation a column vector with the coefficients, while each cell of `supp` has a matrix with the corresponding support for these coefficients.

Code 1.9. 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);
```

After the definition of the system, it is possible to retrieve the most important information about the system via the overloaded `disp` and `probleminfo`, or directly via dot indexing.

Code 1.10. You can get more information about a problem via `disp`:

```
>> disp(system)

    system of multivariate polynomial equations in the
    standard monomial basis:
        - number of equations = 3
        - number of variables = 3
        - maximum total degree = 4
```

The information can be accessed via `probleminfo` or directly via dot indexing:

```
>> [n,s,di,dmax,nnze] = probleminfo(system);
>> eqs = system.coef;
```

In order to create a random dense system, `randomsystem(n,s,dmax)` 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`.

1.2.2 Rectangular multiparameter eigenvalue problems

Similarly, a rectangular MEP can be represented by a cell array that contains all the coefficient matrices in the correct monomial ordering (including zero matrices). Now, every entry corresponds to one coefficient matrix, ordered in a particular monomial ordering. 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, together with the total degree and number of variables of the problem, is then given to the `mepstruct` constructor.

Code 1.11. By combining the different coefficient matrices, we can construct a linear two-parameter eigenvalue problem. Contrary to `systemstruct`, `mepstruct` requires information about the maximum total degree `dmax` and number of eigenvalue `n`.

```
>> dmax = 1; n = 2;
>> A00 = randn(4,3); A10 = randn(4,3); A01 = randn(4,3);
>> mat = {A00, A10, A01};
>> mep = mepstruct(mat,dmax,n);
```

Again, you could decide to enter the problem directly in its internal representation. A single rectangular MEP consists out 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 1.12. A MEP can also be constructed directly by giving the internal representation to `problemstruct`.

```
>> coef = {mepshape(mat)};
>> supp = {[0 0; 1 0; 0 1]};
>> mep = problemstruct(coef,supp);
```

It is possible to retrieve the most important information about the MEP via the overloaded `disp` and `probleminfo`, or directly via dot indexing. Quickly constructing a rectangular MEP with random coefficient matrices is also very easy with `randommep(n,k,l,dmax)`, where `n` is the number of eigenvalues, `k` and `l` are the number of rows and columns of the coefficient matrices, and `dmax` is the maximum total degree.

1.2.3 Database with test problems

The accompanying database contains many test problems that can be used directly with the functions in MacaulayLab. More information about a system of multivariate polynomials or rectangular MEP can be obtained via `help problem` or `disp(problem)`. Most problems in the database already contain information about the number of affine solutions, total number of solutions, required time to solve the system, etc.

Code 1.13. `help(problem)` and `disp(problem)` give more information about the `problem`.

```
>> help noon3;

noon3 contains a system of multivariate polynomial
equations.

    [system] = noon3() returns the system of multivariate
    polynomial equations.

>> disp(noon3)

    system of multivariate polynomial equations in the
    Vandermonde basis:
        - number of equations = 3
        - number of variables = 3
        - maximum total degree = 3
```

Some problems have one or multiple additional parameters.

Code 1.14. The MEP `arma11` requires a vector as parameter.

```
>> y = randn(10,1);
>> example = arma11(y);
```

1.3 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 different problems. We give a step-by-step solution approach (Section 1.3.1), but the user is most likely to use the standard solution approach (Section 1.3.2).

1.3.1 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 system of multivariate polynomial equations or block Macaulay matrix for a rectangular MEP. The function `macaulay(problem,d)` builds the (block) Macaulay matrix of degree `d` that incorporates the `problem`.

Code 1.15. 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 standard (block) Macaulay matrix solution approach in this toolbox uses a basis matrix of the null space the (block) Macaulay matrix (i.e., a basis matrix contains the basis vectors of the column space as its columns.). This matrix can be computed directly via the standard approach (a singular value decomposition) `null`, via the recursive approach `macaulayupdate` and `nullrecrmacualay`, or via the sparse approach `nullsparsemacaulay`.

Code 1.16. The standard approach to compute a basis matrix of the 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 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
```

Once you have a basis matrix of the null space, you want to look for the standard monomials and determine the gap zone (which also gives the number of affine solutions of the problem). Of course, one can also determine the gap directly via `gap`.

Code 1.17. 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`:

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

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

Code 1.18. You can perform a column compression via:

```
>> nrows = nbmonomials(dgap,n);
>> [~,~,Q] = svd(Z(1:nrows,:));
>> V = Z*Q;
>> W11 = V(1:nrows,1:ma);
```

There is also a direct function to obtain $\boldsymbol{W}_{11}$.

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

Solving the problem can be done via shifting the null space. `shiftnullspace` constructs multiple shift problems in the null space that yield the solutions of the problem.

Code 1.19. The function `shiftnullspace(W,shift,rows,l)` considers n shift problems in the basis matrix `W`. `rows` indicates the rows that are shifted and `l` is the length of the eigenvector.

```
>> D = shiftnullspace(W,shift,rows,l);
```

When `rows = NaN`, all degree blocks up to the gap zone are shifted.

1.3.2 Standard solution approach

Of course, there is a solver implemented that takes care of all these intermediate steps and checks whether the 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 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 1.20. Two examples of using `macaulaylabproblem`:

```
>> roots = macaulaylab(redeco6);
>> eigentuples = macaulaylab(hkp2);
```

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

Code 1.21. By using `macaulaylab(problem,dend)`, the maximum degree of the (block) Macaulay matrix is set to `dend`.

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

The solver has many options and outputs a lot of information, which can be set when one choses to do this. Table 1.1 gives an overview of the different options, but the default options should result in the fastest and most stable execution. The options can be set via supplementing a options structure to the solver. For example, asking the solver for a verbose output of its execution can be achieved by setting `options.verbose = true`.

Code 1.22. A structure can be used to set the options of the algorithms.

```
>> options = struct;
>> options.verbose = true;
```

These options are supplemented to `macaulaylab(problem,dend,options)`:

```
>> [X, output] = macaulaylab(noon5,15,options);

                        MacaulayLab
-------------------------------------------------------------
The system has 5 equations in 5 variables (maximum total
degree is 3). The selected polynomial basis is the standard
monomial basis.

The solvers tackles the system via the null space of the
Macaulay matrix:
    * Building a basis matrix of the null space (sparse -
    block row-wise)
        ⋮
The solver results in 233 affine solution candidates in
3.5888 seconds.
        accuracy before clustering = 8.8622e-11
        accuracy after clustering = 8.8622e-11
        cluster tolerance = 1.0000e-03
        rank tolerance = 1.0000e-10
-------------------------------------------------------------
```

The output structure contains a lot of additional information:

```
>> output

output =
    struct with fields:
        accuracy: 8.8622e-11
            ⋮
        shiftvalues: [233x1 double]
```

1.4 Monomial ordering and polynomial basis

MacaulayLab is implemented indecently from the monomial ordering and polynomial basis, which means that you can supply a certain monomial ordering or polynomial basis and use all the functions without any adaptations. Choosing a certain monomial ordering is done by giving a function that determines the position of a monomial to the solver, e.g., `posgrinvlex` and `posgrevlex`. The definition of a basis requires the user to supply (or use) two functions:

- Definition of the shift property, e.g., `basismon` and `basischeb`.
- Evaluation of a problem, e.g., `evalmon` and `evalcheb`.

These functions are given to MacaulayLab as function handles.

Code 1.23. It is possible to construct the Macaulay matrix in any polynomial basis or monomial ordering. `basis` and `order` should be two functions that implement the basis multiplication and the position of a monomial in the monomial ordering.

```
>> basis = @basischeb; % Chebyshev polynomial basis
>> order = @ordergrevlex; % grevlex monomial ordering
>> M = macaulay(problem,d,basis,order);
```

The toolbox has some pre-implemented functions for the monomial ordering and polynomial basis. For the former, `posgrinvlex` and `posgrevlex` allow the user to work in the GRINVLEX or graded reverse lexicographic (GREVLEX) ordering. The standard monomial basis (`basismon` and `evalmon`) and Chebyshev basis (`basischeb` and `evalcheb`) are implemented for the latter. The user can also use its own functions and give them to `macaulaylab`:

Code 1.24. You can give a different polynomial basis to the solver:

```
>> options.basis = @basisdefinition;
>> options.eval = @evaldefinition;
>> solutions = macaulaylab(problem,d,options);
```

1.5 Other useful functions

When working with systems of multivariate polynomial equations and rectangular MEPs, some other functions might come in handy:

- `nbmonomials(d,n)` gives the number of multivariate monomials in `n` variables up to degree `d`.

Table 1.1. Overview of the different options that can be used by the MacaulayLab solver. Section 1.4 gives more information about the options to set the monomial ordering and polynomial basis.

option	**type**	**default**	**explanation**
`subspace`	`boolean`	`true`	choice of subspace
`blocked`	`boolean`	`true`	blocked version of the algorithm
`algorithm`	`string`	`'sparse'`	algorithm for the subspace
`tol`	`double`	`10e-10`	tolerance for rank checks
`clustertol`	`double`	`10e-10`	tolerance for clustering
`cluster`	`boolean`	`true`	flag for clustering
`posdim`	`boolean`	`true`	flag for positive-dimensional solution sets
`shiftpoly`	`double(m,n+1)`	`[randn(n,1) eye(n)]`	shift polynomial
`basis`	`function`	`basismon`	definition of the polynomial basis
`position`	`function`	`posgrinvlex`	definition of the monomial ordering

- `monomialsmatrix(d,n)` creates a matrix with as its rows all the different monomial vectors in `n` variables up to degree `d`.
- `bezout(system)` determines the Bézout number of a system of multivariate polynomial equations `system`.
- `kushnirenko(system)` determines the Kushnirenko bound on the number of affine solutions a `system` of multivariate polynomial equations.
- `bkk(system)` determines, for a `system` of multivariate polynomial equations, the Bernstein–Khovanskii–Kushnirenko (BKK) bound on the number of affine solutions.
- `hkp(mep)` gives the Hochstenbach–Košir–Plestenjak (HKP) bound on the number of solutions for a rectangular `mep`.