New package for effective polynomial computation in MATHEMATICA

Petr Kujan
Department of Control Engineering
Faculty of Electrical Engineering
Czech Technical University in Prague
Prague, Czech Republic
e-mail: kujanp@fel.cvut.cz

Martin Hromčík, Michael Šebek
Centre for Applied Cybernetics
Faculty of Electrical Engineering
Czech Technical University in Prague
Prague, Czech Republic
fax: +420-2-2435 7681
e-mail: m.hromcik@c-a-k.cz

***Abstract*— This report describes our work on implementation of effective numerical routines for polynomials and polynomial matrices in the MATHEMATICA software. Such operations are recalled during the controller design process if the so called polynomial or algebraic design methods are employed. This research is also motivated by the fact that MATHEMATICA developers pay attention to control engineers needs and produce the Control Systems Professional package for use with MATHEMATICA and, as we believe, a set of routines for algebraic approach could conveniently complement the existing bunch of programs primarily intended for state-space representations.**

I. INTRODUCTION

Speaking in broad terms, we can distinguish three main approaches to analysis and design of linear control systems.

The classical frequency-domain methods have evolved from the analysis of frequency responses of linear dynamics systems. Their main formal mathematical tool is the theory of functions of complex variable, in particular the Laplace transform in case of continuous-time and the Z-transform for the discrete-time systems. Systems are described in terms of their transfer functions reflecting just the external input-output relations, which brings about some difficulties related to the internal stability of the closed loop and to the realization of the compensator. The used formalism also causes that the domain of classical methods is reduced to SISO time-invariant linear systems. But despite these limitations, the classical methods still remain very popular, namely in the community of practising engineers, for their simplicity and effectivity in many control problems encountered in industry.

The classical methods are suitable for computational processing and a lot of software tools supporting the design process are available.

The drawbacks of the classical approach and the increasing complexity of systems to be controlled resulted in new methods of synthesis, usually called the state-space or modern approach. The methods rely upon the exact definition of the state that is systematically used both for the deeper analysis of the plant (the state provides the insight into the internal structure of the system) and for the synthesis of the compensator (the knowledge of the state is employed for compensation). The main formal tools are differential equations, vector spaces and matrix theory. The modern methods are applicable to much wider class of situations than the classical ones, e.g. to MIMO and time-varying systems. However, they have not become so popular, namely among the practicing engineers, for the necessity of finding the state-space model and for the need of state reconstruction in case it cannot be directly measured.

From the numerical point of view, the state-space design methods for linear systems rely on numerical linear algebra which is a powerful tool. Since the 50s a lot of effort has been devoted to the development of accurate and numerically stable algorithms for linear algebra problems encountered in a large number of scientific computations.

The origin of the polynomial or algebraic approach is dated to the early 70s. The polynomial matrices forming polynomial matrix fractions are introduced to handle MIMO cases. Systems are described by input-output relations, however the transfer functions are not regarded as functions of complex variable but as algebraic objects. The design procedure is then reduced to algebraic operations with polynomial matrices, typically to solving algebraic polynomial equations. This approach not only enables to resolve many existing control problems in a more elegant and unifying way but also provides further insight into the structure of the control systems and shows new relationships between various control tasks.

The polynomial methods involve rather unusual mathematics which brings about some difficulties with numerical processing. While the numerically reliable algorithms for operations with constant matrices are standard and programmed in many computing packages at present, it is not true in the case of matrices with polynomial entries. Many textbooks on algebraic design methods contain some ideas how the computations could be performed, but they usually rely on elementary operations and are not numerically stable. In journals on control and on scientific computation only a small amount of isolated papers dealing with this problematic could be found until recently.

In recent years we are the witnesses of a breakthrough in the field. New class of reliable numerical algorithms have finally been published and the methods have been systematically studied from the point of numerical properties. This investigation has also given rise to the first practically applicable software for numerical operations with polynomial matrices,

the Polynomial Toolbox for MATLAB.

Having verified proper performance of new numerical routines by programming the Polynomial Toolbox for MATLAB, we realized that MATLAB is not the only software used for control systems design. MATHEMATICA also provides a reliable computational environment and a comprehensive package for control engineers, the Control Systems Professional, and also supports symbolic computations, we decided to implement some useful polynomial routines in this software. Our first results are presented in this paper.

The paper is organized in the following manner. First, the MATHEMATICA system is introduced in brief in section II. Description of our polynomial package then follows in section III. Particular subsections are devoted to polynomial objects definitions, basic and advanced functions for algebraic operations with polynomials and polynomial matrices, as well as to interesting implementational issues and illustrative examples. Results of extensive numerical testing of selected routines are summarized in section IV. Performance of our routines in a practical example of model matching problem is presented in section V. Finally, plans for further development of the package are given along with some concluding remarks.

II. SYSTEM MATHEMATICA

MATHEMATICA is a program with fully integrated support for technical computations. Among other features, it is equipped with a powerful symbolic language and is capable to handle a wide class of mathematical objects including polynomials. Nevertheless, practical problems of control systems analysis and design, based on the so called algebraic approach and involving polynomial computations, cannot be practically treated by standard symbolic tools of MATHEMATICA - the computational times become unacceptably high.

For this reason a decision has been made to implement also numerical routines in MATHEMATICA to cope with polynomials and matrices of such objects. Unlike the symbolic procedures, numerical algorithms are fitted to the special structure of polynomial matrices and use effective tools of numerical linear algebra.

III. MATHEMATICA PACKAGE FOR POLYNOMIAL OBJECTS

Naturally it is possible to create additional modules in the MATHEMATICA system extending the basic system functions. A constructed polynomial package can be implemented into the system easily, including help and examples. Standard functions for package operations e.g. `Needs["Poly‘Master‘"]`or shortly `<<Poly‘` are used for upload.

A. New polynomial objects

Functions joining together a polynomial matrix or a polynomial with their variable or their degree were created to improve the polynomial object operation. These basic function form an object definition in the MATHEMATICA system.

Polynomial matrixes could be entered in three fundamental forms using the following constructors:

- **Polynomial Matrix Form:**
  The command `PM[`pm`,`var`]` creates a new object of the class `PM` (Polynomial Matrix). Input arguments are a rectangular matrix (MATHEMATICA list) pm with polynomial entries and its considered variable var.
  Example 1. Square polynomial matrix $\begin{pmatrix} k & s \\ 2-5s & 3 \end{pmatrix}$ in `PM[]` representation.
  ```
  PM[{{k,s},{2-5s,3}},s]
  ```
- **Polynomial Matrix Coefficients:**
  The command `PMC[`pmc`,`var`,`deg`]` creates a new object of the class `PMC` (Polynomial Matrix Coefficients). Input arguments are a list of rectangular matrix with numbers or symbols pmc, its variable var and degree of polynomial matrix deg.
  Example 2. Square polynomial matrix from example 1 in `PMC[]` representation.
  ```
  PMC[{{{k,0},{2,3}},{{0,1},{5,0}}},s,1]
  ```
- **Polynomial List of Coefficients:**
  The command `PLC[`plc`,`var`]` creates a new object of the class `PLC` (Polynomial List of Coefficients). Input arguments are a rectangular matrix with list of scalar polynomial coefficients and its considered variable var.
  Example 3. Square polynomial matrix from first example in `PLC[]` representation.
  ```
  PLC[{{{k,0},{0,1}},{{2,-5},{3,0}}},s]
  ```

Scalar polynomials can be set in two forms similarly:

- **Scalar Polynomial**
  The command `P[`pol`,`var`]` creates a new object of the class `P` (Polynomial). Input arguments are a scalar polynomial and its variable var.
  Example 4. Scalar polynomial $k + 3s + s^2$ in `P[]` representation.
  ```
  P[k+3s+s^2,s]
  ```
- **Polynomial Coefficients**
  The command `PC[`pol`,`var`]` creates a new object of the class `PC` (Polynomial Coefficients). Input arguments are a list of scalar polynomial coefficients and its considered variable var.
  Example 5. Scalar polynomial from example 5 in `PC[]` representation
  ```
  PC[{k,3,1},s,2]
  ```

As needed, particular representations are used in calculations of particular functions.

All entries are automatically converted to these standardized forms of the polynomial objects. A user can enter non-complete objects even, in a form of matrices. A missing variable is found or a value from the global variable `$GlobalVar` is filled.

```
In[1]:=   PM[{{1+z,3},{10z^2,-z}}]
```

```
Out[1]=   PM[{{1+z,3},{10z^2,-z}},z]
```

When a degree is not set, the number of elements in a list is completed.

```
In[2]:=   PC[{k,-2,3}]
Out[2]=   PC[{k,-2,3},s,2]
```

Besides numeric values, the matrices can contain symbols too. The object definition determines that all representations are convertible easily to one another.

```
In[3]:=   PMC[PM[{{k+2s,3+4s}},s]]
Out[3]=   PMC[{{{k,3}},{{2,4}}},s,1]
```

B. Implementation in MATHEMATICA

The objects are defined as condition pattern of functional arguments. Some tests then follow if input arguments correspond to the polynomial object (polynomial matrix, polynomial, coefficients of the polynomial matrix, coefficients of the polynomial) in the range of a given variable. In case of correct values of the object the condition is evaluated as `False` and then a correct input values is returned on the functional output. When the arguments do not meet the given object (the matrix is not polynomial), then the condition is evaluated as `True` and the value `$Failed` is returned (i.e. the input does not meet the definition of the object). The definition of the object is given by several other functions which convert the input into a required form and this one is tested as the condition pattern according to the previous object finally. This fact allows a simplified entry of the given object. For instance, it is not needed to entry a variable of the polynomial matrix explicitly or a vector can be entered in place of the matrix. The last definitions of the objects catch all incorrect entries of the object and return the value `$Failed`. The principle of a gradual testing of arguments of the same function in the MATHEMATICA system is applied here.

Because of the testing of the input polynomial matrix is time consuming, a global logical variable (also condition pattern) is introduced, which have set the `False` value in the functions operating on polynomial objects. Though, the time consuming tests verifying, if it is a polynomial object, will not be performed in the function body.

The definition of the object contains defined functions for the conversions of the particular polynomial objects furthermore.

Preview of MATHEMATICA *code for definition* `PM` *object.*

```
(* master definition with test if
   elements are scalar polynomials  *)
PM[A_?MatrixQ, var_Symbol]/; TestPM /;(
    If[ PMTest[A,var],
      False,Message[General::notpm, A];
      True
    ]
  ) := $Failed;

(* another possible definitions *)
PM[A_?MatrixQ]/; TestPM :=
    PM[A,GetVariable[A]];
PM[v:_?VectorQ]/; TestPM :=
    PM[{v},GetVariable[{v}]];

(* further functions for conversion
   among polynomial objects         *)
PM[A:pmc[pmcA, var, deg]] :=
Module[{n = Length[pmcA], pmA},
   TestPM = False;
   pmA = PM[Plus @@
      MapIndexed[
         #1*var^(deg - n + First[#2])&,
         pmcA
      ], var];
   TestPM = True;
   Return[pmA];
];
PM[A:plc[plcA, var]] := Module[..];
PM[A:pm[pmA,var]] := A;

(** other definition are $Failed **)
PM[_] := $Failed;
PM[_,_,__] := $Failed;
```

C. Implemented functions

The routines implemented in MATHEMATICA within our new polynomial package are namely the following:

1) **Polynomial matrix determinant :** For the polynomial matrix with numerical coefficients the implemented algorithm is adopted from [2]. It is the most efficient published method for polynomial matrix determinant computation. Non-numerical polynomial matrices are resolved by standard MATHEMATICA function for symbolic determinant.

Implementation: MATHEMATICA *pseudocode*

Step 1 Redefine standard function `Det[]` for polynomial matrix object.

```
Unprotect[Det];
Det[A:(pm[pmA,var]|plc[plcA,var])
  ,opts___?OptionQ]/;
  (SquareQ[A] && NumericQ[A] )
 := detpm[PM[A],opts];
```

Step 2 Compute the upper bound for the degree of the determinant:

```
degofdet =
 Min[Plus@@coldegA,Plus@@rowdegA];
```

Step 3 Using Fourier function:

```
samples = Transpose[
 Map[Fourier[#]&,CoefficientList[
  pmA+0.var^dgofdt,var],{2}
],{2, 3, 1}];
```

Step 4 Compute standard determinant:

```
dets = Det /@ samples;
```

Step 5 Using inverse Fourier:

```
sol = InverseFourier[
     dets,FourierParam->{1,-1}];
PC[sol,var,dgofdt]//Return;
```

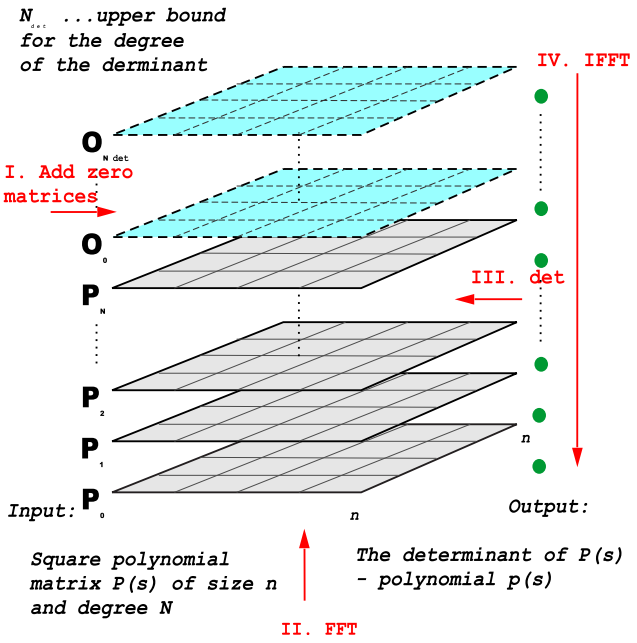

Fig. 1. Computational scheme of the algorithm [2].

Example 6. The command `A = PMRandom[`deg, I, J`]` generates a random I-by-J polynomial matrix A of degree deg with normally distributed coefficients. If J is missing then a square I-by-I matrix is created.

```
In[4]:=   A = PMRandom[2,2]; Det[A]
Out[4]=   P[-68.+49.s-39.sˆ2+32.sˆ4,s]
```

2) **Multiplication of polynomial matrices:** Multiplication of polynomial matrices can be rephrased in terms of constants as a product of Sylvester matrices and column coefficient matrices related to the inputs. For details see [6].

Implementation: MATHEMATICA *pseudocode*

Step 1 Function for computation of Sylvestr matrix.

```
SylvestrMattrix[A:pmc[pmcA,var,deg],
                col_?Positive] :=
  Module[{m, n, k, mat},
    {m,n} = Dimensions[A];
    mat = Table[0,{m*(col+deg)},
                {col*n}];
    Do[If[ (i>=j && i<=j+deg),
        mat[[
          Range[(i-1)*m+1,i*m],
          Range[(j-1)*n+1,j*n]
        ]] =  pmcA[[i-j+1]]
       ],{i, deg+col}, {j, col}];
    mat//Return
  ];
```

Step 2 Redefine standard function `Dot[]` for polynomial matrix object.

```
Unprotect[Dot];
Dot[A:(pm[pmA,varA]|pmc[pmcA,..]),
    B:pm[pmB,varB]] /;
 (varA===varB &&
  Dimensions[A][[2]]==Dim[B][[1]]
 ):= PM[dotpmc[PMC[A],PMC[B]]];
```

Step 3 Function `dotpmc[]` compute multiplication of Sylvest matrices and make from it the final polynomial matrix.

```
{{rA,cA},{rB,cB}} =
   {Dimensions[A],Dimensions[B]};
dots = Dot[
         SylvestrMatrix[A,degB+1],
         SylvestrMatrix[B,1]
       ];
PMC[Partition[dots, rA], varA,
    degA+degB]
```

Example 7.

```
In[5]:=
   b = PMRandom[{3,2}];
   {a,c} = Table[PMRandom[{2,3}],{2}];
   a.c.b
Out[5]=   PM[{{1+3sˆ2,..}},s]
```

3) **Linear matrix Diophantine equation** $A.X = B$**:** Adopted from [3]. Similarly to the case above, the linear equation of the given type can be reformulated in terms of Sylvester matrices. The essential problem lies in the determination of the result's degree which is performed via a weighted binary search process in the program.

Implementation: MATHEMATICA *pseudocode*

Step 1 Estimation of lower and upper bounds of X's degree: `mindeg`, `maxdeg`

Step 2 Binary search for minimum degree.

```
degree =
  mindeg+Floor[(maxdeg-mindeg)/2];
While[maxdeg-1 =!= mindeg,
 coeA=SylvestrMatrix[A,degree];
 coeB=Join[ pmcB,
        ZeroMatrix[
          (1+degA+degree)*rA
          -1+degB)*rB,cB ];
 sol=LinearSolve[As[[1]],Bs[[1]]];
 If[solution exist,(* then *)
   mindeg = degree,(* else *)
   AppendTo[listsol, sol];
   maxdeg = degree;
 ];
 degree =
  mindeg+Floor[(maxdeg-mindeg)/2];
]
```

4) **Linear matrix Diophantine eq.** $A.X + B.Y + .. = C$**:** This equation can be written as $[AB\cdots].[XY\cdots]^T$. Such a

way, the problem is conversed to the linear equation of the type $A.X = B$ which is discussed above. Basically, the numerical computation is based on the program for the previous case.

Example 8. Polynomial matrix b and polynomial marix in argument function `DESolve[`eq`,`$\{vars\}$`]` are with parametrs n, m.

```
In[6]:=
  b = PM[{{n s,1-s^2,3},{5s,1,n-s}},s];
  d = PMRandom[2,3,3];
  DESolve[
    PM[{{1,m},{-2+s,3}},s].x+b.y-d==0,
    {x, y}
  ]
Out[6]= {x->PM[{{-2+s/n..}},s],
         y->PM[{{5-s..}},s]}
```

5) **Bilateral symmetric scalar polynomial equation of type** $a' * x + x * a' = b$**:** Solves the equation with the Sylvester matrix method. For details see [4].

Implementation: MATHEMATICA *pseudocode*

Step 1 Preparation input coefficients of scalar polynomial for Sylvestr matrix computation.

```
lb = PadLeft[pcB,degB+1,0];
If[degA<=degB/2,
(* then *)
  lb=Partition[lb,1,2];
  k=PadRight[
      PadLeft[pcA,degA+1,0],
      {degB+1}];
  i=degB/2+1;
  j=degB-degA+1,
(* else *)
  lb = PadRight[
         Partition[lb,1,2],
         {degA},{{0}}];
  k = PadRight[
        PadLeft[pcA,degA+1,0],
        {2*degA-1}];
  i = degA;  j = degA; ];
```

Step 2 Create modify Sylvestr matrix for scalar polynomial.

```
mls = Table[0, {i}, {j}];
Do[ If[jj<=2*ii-1, If[EvenQ[jj],
   mls[[ii,jj]]=-2k[[2*ii-jj]],
   mls[[ii,jj]]=2k[[2*ii-jj]]
]],{ii, i},{jj, j} ];
```

Step 3 Compute set of linear equations.

```
sol = LinearSolve[mls, lb];
PC[Flatten[sol],var,j-1]//Return
```

Example 9.

```
In[7]:=  a=PC[{1, 2, 3}];b=P[1 + 2s^2];
         AXXABSolve[a,b]
Out[7]=  P[1/2+1/4s,s]
```

D. Advantages of polynomial objects

Introduction of particular polynomial objects as described above brings several benefits for effective implementation of routines working on polynomials and polynomial matrices.

First, standard MATHEMATICA functions can be easily redefined for our new `PM`, `PMC` and other objects. The functions in concern include matrix-matrix addition, multiplying by a constant, matrix-matrix multiplication or a determinant calculation. The redefinition is carried out by means of a command `Unprotect[std_function]`. Then it is allowed to modify a standard function directly, optionally it can be referred to its own function which does a faster calculation with the polynomial object.

In addition, testing whether an input argument is of the desired type can be reduced to one simple check - usually the first step in a function performs a trivial comparison between its input's pattern and related patterns for particular polynomial objects. Moreover, once this test is passed it is easy to refer to e.g. a matrix or a variable in the body of the function. See the following example of the determinant solver code:

IV. RESULTS

As expected, the implemented methods specifically tailored for polynomial objects are considerably faster than the general symbolic routines incorporated in MATHEMATICA.

1) ***Determinat of polynomial matrix.***
Square test matrices of size n and degree N have random real coefficients.

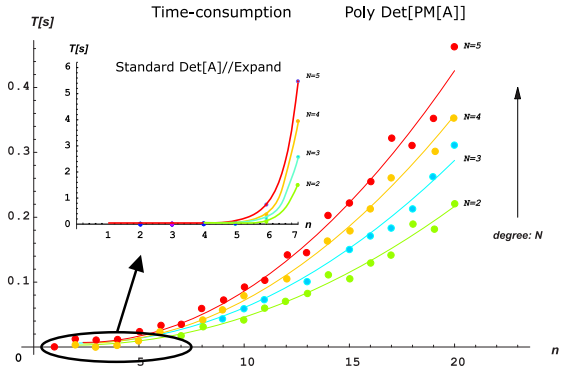

Fig. 2. Time - consumption of the function `Det[`pol`]` and standard MATHEMATICA function `Det[`mat`]` for matrix.

TABLE I
TIME - CONSUMPTION.

Input sq. mat. n	N	Poly `Det[A,B]`	Standard `Det[a,b]//Expand`
6	2	0.02	0.120
6	3	0.03	0.251
7	5	0.05	5.418
9	5	0.100	395
20	5	0.431	*

A star (*) in tables means that the execution time exceed 1000 seconds.

2) ***Multiplication of polynomial matrices.***

TABLE II

TIME - CONSUMPTION.

Input sq. mat. A,B n	N_A	N_B	Poly `Dot[A,B]`	Standard `Dot[a,b]//Expand`
10	10	3	0.07	0.17
10	10	5	0.12	0.43
50	20	20	6.05	*

V. PRACTICAL EXAMPLE

Let us consider the model matching problem according to the figure 3. In the depicted setup the plant $\mathbf{B}/\mathbf{A}$ is given. The design task is to compute the polynomials $\mathbf{M}$,$\mathbf{N}$ and $\mathbf{T}$ such that the overall closed -loop transfer function is equal to a prescribed $\mathbf{F_m}$.

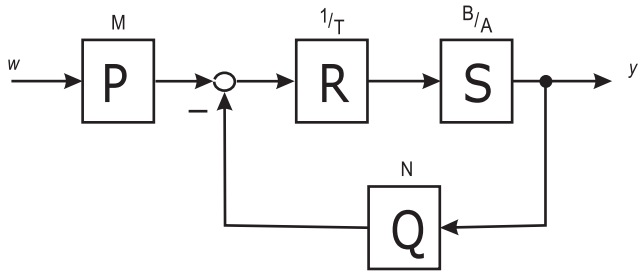

Fig. 3. Schema for matching problem.

Let the plant be given as

$$\mathbf{S} = \frac{\mathbf{B}}{\mathbf{A}} = \frac{1}{s(s+1)(s+4)}.$$

The plant transient is not sufficiently fast and for this reason we will try to alter its dynamics according to the model

$$\mathbf{F_m} = \frac{\mathbf{D}}{\mathbf{C}} = \frac{\mathbf{10^3}}{(\mathbf{s+10})^\mathbf{3}} = \frac{\mathbf{10^3}(\mathbf{s+100})^\mathbf{3}}{(\mathbf{s+10})^\mathbf{3}(\mathbf{s+100})^\mathbf{3}}$$

using the depicted feedback configuration.

The overall closed-loop function reads

$$\mathbf{G} = \frac{\mathbf{BM}}{\mathbf{BN+AT}}.$$

By equating numerators and denominators we arrive at the following equations:

$\mathbf{BM} = \mathbf{D} \rightarrow \mathbf{M} = 10^3(s+100)^2$

$\mathbf{BN}+\mathbf{AT} = \mathbf{C} \rightarrow \mathbf{N}+(s+1)(s+4)s\cdot\mathbf{T} = (s+10)^3(s+100)^2$

Last equation is solved by the command `DESolve[]`

```
In[8]:=   DESolve[
    N+s(s+1)(s+4)T==(s+3)^3(s+100)^2,
    {N,T}]
Out[8]=
  {T -> P[15171+225s+s^2,
   N -> 10.10^6+3139316s+284245s^2}
```

VI. FURTHER DEVELOPMENT

So far, convenient objects for polynomial or polynomial matrix representation have been defined and some reliable routines for basic algebraic operations over polynomial matrices have been implemented along with determinant computation and one-sided Diophantine equation solver. Basically, all problems listed are linear and can be used satisfactorily to design controllers based on linear operations, including pole-placement and deadbeat compensators. On the other hand, most practically useful control strategies, including H2, H infinity and l1 control require some additional quadratic operations. In the phrases of polynomial approach, these are represented by spectral factorization and plus-minus factorization for instance. These routines are necessarily based on numerical iterative schemes and can by no means be accomplished by any sort of symbolic manipulations. Quite naturally, the various polynomial factorization problems are of interest in the future. The first step towards the implementation of a reliable spectral factorization algorithm, see [7], has been done already by programming the symmetric Diophantine equation solver.

VII. CONCLUSION

Implementation of reliable algorithms for polynomials and polynomial matrices in MATHEMATICA with respect to algebraic theory of control systems design has been the subject of this report. After a review of polynomial design methods the basic structure of developed package was enlightened and implemented functions were introduced. Apart from their description, attention was also paid top interesting implementational issues. Finally, performance of programmed functions was demonstrated by means of numerical experiments and a practical control example. The package as described in this paper is not considered complete. Possible directions of further development have been also presented and advocated.

ACKNOWLEDGMENT

The work of M. Hromčík and M. Šebek has been supported by the Ministry of Education of the Czech Republic under contract No. LN00B096.

REFERENCES

[1] T.B. Bahner, MATHEMATICA *for Scientists and Engineers*, Addison-Wesley Publishing Company, Inc., 1995.

[2] D.B. Wagner, *Power programming with* MATHEMATICA*: the Kernel*, McGraw - Hill, 1996.

[3] M. Hromčík, M. Šebek, *New Algorithm for Polynomial Matrix Determinant Based on FFT, Proceedings of the European Control Conference ECC'99*, Karlsruhe, Germany, 1999.

[4] D. Henrion, *Reliable Algorithms for Polynomial Matrices, PhD. Thesis*, Institute of Information Theory and Automation, Czech Academy of Sciences, Prague, 1998.

[5] H. Kwakernaak, M. Šebek, *PolyX Home Page*, `http://www.polyx.com/`

[6] *The Polynomial Toolbox for* MATLAB *Manual*, PolyX `http://www.polyx.com/`.

[7] J. Ježek, V. Kučera, *Efficient Algorithm for Matrix Spectral Factorization*, Automatica, vol. 29, pp. 663-669, 1985.

[8] I. Bakshee, MATHEMATICA - *Control system professional*, Wolfram research, Inc., 1996.

[9] V. Kučera, *Analysis and design of discrete linear control systems*, Czechoslovak academy of sciences, Academia, Prague, 1991