NICONET Report 1999-3

Efficient implementation of Subspace method identification Algorithms [1]

B.R.J. Haverkamp[2]

March 1999

[1]This paper presents research results of the European Community BRITE-EURAM III Thematic Networks Programme NICONET and is distributed by the Working Group on Software WGS. *WGS secretariat*: Mrs. Ida Tassens, ESAT - Katholieke Universiteit Leuven, K. Mercierlaan 94, 3001-Leuven-Heverlee, BELGIUM. This report is also available by anonymous ftp from *wgs.esat.kuleuven.ac.be* in the directory /pub/WGS/REPORTS/ as file name nic1999-3.ps.Z
[2]Faculty of Information Systems and Technology, Delft University of technology, P.O. Box 5031, NL-2600 GA Delft, The Netherlands

Abstract

This paper summarises the results of a study to improve existing Subspace Method Identification (SMI) algorithms. Significant improvements in calculation speed can be achieved by combining components from existing algorithms namely N4SID and MOESP. A second improvement can be achieved by more efficient implementation of critical parts of the algorithms.

1 Introduction

Subspace method identification(SMI) is a fairly new branch of the identification tree. Early references about SMI date back to the early nineties[1, 2, 3]. In the last ten years, SMI has become a mature tool for the identification of multi-variable linear systems and special cases of nonlinear systems. In many fields it is applied successfully [4, 5].

Many different implementations of SMI exist. these implementations differ significantly, theoretically as well as in practical implementation. In [6] a comparative study between the three most commonly used algorithms CVA, N4SID and MOESP was made. The algorithms were compared on the aspects of computational complexity, prediction error and simulation error.

The main conclusion of this study was that the three algorithms differed much in computational complexity. The CVA method showed a significantly lower number of floating point operations (FLOPS) compared to the other two methods. A second study was then proposed to find the cause of this difference, and to examine possible reductions in computational needs for the MOESP and N4SID.

In this article the results are summarised of a study to reduce the computational complexity of the the various types of subspace identification methods and how the calculation speed can be increased.

First we have examined the MOESP and N4SID algorithms and divided them into modules. These modules are separate independent parts of the algorithm. By combining the fastest modules from the two algorithms, we can improve the overall computational speed. Secondly we looked at possible ways to speed up the calculation futher by implementing certain modules more effectively. This gave a significant increase in speed of the algorithm.

For the implementation of the different algorithms and combinations of them, we have used a tool called MWEB. MWEB allows to construct a single m-file from different modules. These modules consist of normal MATLAB code plus a LaTeXdescription of the MATLAB code. Modules can also include other modules. In this way we can easily construct the mixed algorithms. By combining MATLAB code and LaTeXdocumentation are able to write very structured and readable programmes.

In the next section, we give a short introduction to MWEB. It explains the use of MWEB and the benefits of it. We also give a little example to demonstrate the use of MWEB. In section 3, we discuss N4SID and MOESP and the division into modules. In section 4, possible combinations of the two algorithms are discussed. In section 5, several ways to speed up the algorithms are discussed. Finally in section 6 some results are given.

2 Introduction to MWEB

This section provides an introduction into the use of MWEB. More detailed information can be found in [7], [8] and [9].

By using MWEB we combine MATLAB and LaTeX. Both the MATLAB and LaTeXcode are stored in a structured way in a WEB file which has the extension `.web`. Processing this file with the program `mtangle` produces a MATLAB file (extension `.m`). Processing the WEB file with the program `mweave` produces a TeX file (extension `.tex`). The `webfiles` package [9] provides a way to include the TeX files produced by `mweave` in a LaTeX document. In the preamble of the LaTeX document the `webfiles` package is loaded as follows:

```
\usepackage{webfiles}
```

The TeXfile is included with the following command:

`\webfile{`*filename*`}`

where *filename* is the name of the TeX file.

A WEB file consists of several sections, that are called *modules*. Each module contains two parts:

1. A LaTeXpart, containing explanatory material, formulae etc.
2. A MATLAB part, containing a piece of program code.

The LaTeXpart must always be present, while the MATLAB part can be empty. The MATLAB part of the first module starts with `@(` followed by the name of a MATLAB file. The matlab parts of the next modules start with `@<` followed by the name of the module, or with `@(` to start a new MATLAB file. In a single WEB file multiple m-files can be programmed.

The following example shows a little program written in MWEB First the raw text is shown. After that, the output of mweave and mtangle are given.

WEB-file

```
@*1 Pythagoras.

In this little example we calculate pythagoras formula
$z=\sqrt{x^2+y^2}$, but taking care not to overflow.
In this function we avoid the explicite
calculation of  $x^2+y^2$ which can overflow in a finite precision
calculation.

@(pythagoras.m@>=
function z=pythagoras(x,y)
@<Find maximum and minimum of x and y@>
@<Calculate z@>
```

```
@* Find maximum and minimum of $x$ and $y$.
Here we find the maximum of the absolute values of $x$ and $y$, and
the minimum of the two. Those values are used in the next module.

@<Find maximum and minimum of x and y@>=
w=max(abs(x),abs(y));
v=min(abs(x),abs(y));

@* Calculate $z$.
Here we calculate $z= w\sqrt{d^2+1}$ with $d=\frac{v}{w}$.

@<Calculate z@>=
if v==0
  z=w;
else
  d=v/w;
  z=w*sqrt(d^2+1);
end
```

1. Pythagoras.

In this little example we calculate pythagoras formula $z = \sqrt{x^2 + y^2}$, but taking care not to overflow. In this function we avoid the explicite calculation of $x^2 + y^2$ which can overflow in a finite precision calculation.

⟨ pythagoras.m 1 ⟩ ≡
  function $z = pythagoras(x, y)$
    ⟨ Find maximum and minimum of x and y 2 ⟩
    ⟨ Calculate z 3 ⟩

2. Find maximum and minimum of x and y. Here we find the maximum of the absolute values of x and y, and the minimum of the two. Those values are used in the next module.

⟨ Find maximum and minimum of x and y 2 ⟩ ≡
  $w = max(abs(x), abs(y))$;
  $v = min(abs(x), abs(y))$;

This code is used in section 1.

3. Calculate z. Here we calculate $z = w\sqrt{d^2 + 1}$ with $d = \frac{v}{w}$.

⟨ Calculate z 3 ⟩ ≡
  if $v \equiv 0$
    $z = w$;
  else
    $d = v/w$;
    $z = w*sqrt(d\hat{}2 + 1)$;

end

This code is used in section 1.

Index of `pythagoras`

List of Refinements in `pythagoras`

m-file

```
function z = pythagoras(x,y)
 w = max(abs(x),abs(y));
v = min(abs(x),abs(y));


 if v==0
  z = w;
else
  d = v/w;
  z = w * sqrt(d^2+1);
end
```

3 Combining MOESP and N4SID

In this section we decompose N4SID and MOESP into several separate modules. These modules are as independent as possible from each other, and have separate functions within the algorithm. Using these module, we then derive a faster algorithm by combining modules from N4SID and MOESP.

1. The MOESP algorithm contains the following modules

1. First the Hankel matrices U_f, Y_f, U_p and Y_p are constructed from the input and output sequences. Define

$$Y_{i,j,N} \quad = \quad \begin{bmatrix} y(i) & y(i+1) & \cdots & y(i+N-1) \\ y(i+1) & y(i+2) & & y(i+N) \\ \vdots & & & \vdots \\ y(i+j-1) & y(i+1) & & y(i+j-N-2) \end{bmatrix} \tag{1}$$

   then $Y_f = Y_{i+1,i,N}$, $U_f = U_{i+1,i,N}$,$Y_p = Y_{1,i,N}$ and $U_p = U_{1,i,N}$ With this the matrix

$$\begin{bmatrix} U_f \\ U_p \\ Y_p \\ Y_f \end{bmatrix}$$

   is constructed, and an RQ factorisation is performed on this matrix.

2. In the second module, a submatrix of the R matrix from the previous module is taken, and an SVD is computed to obtain the extended observability matrix Γ_i.

$$\Gamma_i = \begin{bmatrix} C \\ CA \\ \vdots \\ CA^{i-1} \end{bmatrix}$$

3. MOESP estimates the matrices A and C directly from Γ_i. This is done in the third module.

4. For the calculation of the Kalman gain a least squares problem is solved. Two matrices Lhs, and Rhs, the lefthand-side, and the righthand-side of the equation are constructed. These matrices are identical to the ones used in N4SID, but due to the difference in R, they are found in a different way.

5. The actual calculation of K based on Lhs and Rhs, constructed in the previous module, is done in this module.

6. In the last modules, a least squares estimate of B and D is computed.

This is the MWEB code for the above algorithm. Every bracketed expression, is a module, which consists of a block of matlab instructions or other modules.

⟨`moesp.m` 1⟩ ≡
  function $[A, B, C, D, K] = moesp(u,\ y,\ s,\ n)$;
    ⟨Initialisation 7⟩
    ⟨Construct R for po-moesp 8⟩
    ⟨Find Gamma for po-moesp 9⟩

⟨ Find A and C for po-moesp 10 ⟩
if *nargout* > 4
  ⟨ Construct Lhs and Rhs for po-moesp 11 ⟩
  ⟨ Find K for po-moesp 12 ⟩
end
⟨ Find B and D for po-moesp 13 ⟩

2. The N4SID algorithm can also be decomposed in modules. The decomposition is very similar to the way we treated MOESP in the previous section.

1. In this module the Hankel matrices are constructed from the input and output sequences, and an RQ factorisation is performed on the following matrix:

$$\begin{bmatrix} U_p \\ U_f \\ Y_p \\ Yf \end{bmatrix}$$

   Note that this matrix is slightly different from the one used in MOESP.

2. In the second modules Γ_i is recovered, by an SVD on the appropriate matrices.
3. For the calculation of A, C and the Kalman gain K a least squares problem is solved. The two matrices involved, Lhs and Rhs, the lefthand-side, and the righthand-side of the equation are constructed in this module.
4. In this module the matrices A and C are calculated based on Lhs and Rhs
5. In the fifth module K is recovered from the residuals of the least squares solution of the previous step.
6. With A and C estimated, we can recompute Γ_i based on these estimates. Then the estimated states are recomputed using this new Γ_i.
7. In the last modules, the matrices B and D are found, using Γ_i and the estimated states.

⟨ `n4sid.m` 2 ⟩ ≡
 function $[A, B, C, D, K] = n4sid(u,\ y,\ s,\ n)$;
  ⟨ Initialisation 7 ⟩
  ⟨ Construct R for n4sid 20 ⟩
  ⟨ Find Gamma for n4sid 21 ⟩
  ⟨ Construct Lhs and Rhs for n4sid 22 ⟩
  ⟨ Find A and C for n4sid 23 ⟩
  if *nargout* > 4
   ⟨ Find K for n4sid 25 ⟩
  end
  ⟨ Recompute Gamma 33 ⟩
  ⟨ Construct Lhs and Rhs for n4sid 22 ⟩
  ⟨ Find B and D for n4sid 24 ⟩

3. By combining parts of N4SID and MOESP, it is possible to derive various different algorithms. By combining the faster parts, we hope to find a more efficient variant.

An obvious difference between N4SID and MOESP is the way in which A and C are calculated. In N4SID, the projected states are constructed, after which a least squares solution for A and C is calculated. In MOESP, A and C are found at an earlier stage, directly from the extended observability matrix. Since the latter is faster, we will use this one.

A second difference is the calculation of B and D. In MOESP, a large least squares problem in B, D and possible the initial state is solved. This requires the construction of some large matrices, which cause a noticable increase in computation time. In N4SID, B and D are computed based on matrices obtained during the computation of A and C. This is much faster.

The mixture of N4SID and MOESP we examined is the following: A and C are constructed using the MOESP method, directly from the extended observability matrix. Then B and D are found using N4SID. The module to estimate K needs to use the MOESP variant, since R is built using MOESP, but this is esentially identical to N4SID.

⟨`mix1.m` 3⟩ ≡
  function $[A, B, C, D, K] = mix1\,(u,\ y,\ s,\ n)$;
    ⟨Initialisation 7⟩
    ⟨Construct R for po-moesp 8⟩
    ⟨Find Gamma for po-moesp 9⟩
    ⟨Find A and C for po-moesp 10⟩
    ⟨Construct Lhs and Rhs for po-moesp 11⟩
    if $nargout > 4$
      ⟨Find K for po-moesp 12⟩
    end
    ⟨Find B and D for n4sid 24⟩

4 Further improving the algorithm

In this section we discuss several possible ways to speed up the SMI algorithm. One important time consuming step is the computation of the RQ factorisation. As demonstrated for the SISO case in [10], we can calculate the R factor also by doing a Cholesky factorisation of the correlation of the data matrix. For the computation of this correlation matrix, we can efficiently take advantage of the Hankel structure of the data matrix.

A second possible improvement, in certain circumstances is to calculate the correlations by using the FFT. Especially for large values of the block matrix dimension parameter i and the number of inputs and outputs, this results in an improved performance. This has however not yet been implemented.

Further development is proposed to use fast QR factorisation methods to alternatively compute the R-factor[11, 12].

5 Test results

We tested in total six algorithms, first the 3 original algorithms, MOESP, N4SID and CVA. Next to these the mixed version of MOESP and N4SID, called MIX1 was tested. To examine the speed improvement from the alternative way of computing R, it was incorporate both in an alternative version of MOESP, and in the mixed version, resulting in two new algorithms, with the names FMOESP and FMIX1.

We tested the algorithms first on an artificial random model of order 4, with 2 inputs and 2 outputs. Two sequences of 1000 samples of input/output data were generated using this model. One set was used for identification. the other for validation. The identification output was polluted with 20% measurement noise. The VAF was calculated on the second dataset. To obtain a single number, the VAF was averaged over all outputs. The parameter i was set to 8, and the order of the identified system, n was set to the true order of the system: 4. The experiment was repeated 10 times, after which the results were averaged. These results are shown in table 1.

Secondly, a larger random model of order 10 with 4 inputs and 12 outputs was used to test the algorithms. Again, two sets of 5000 samples of i/o data were generated. The output was polluted with 20% measurement noise. The parameter i was set to 20, and the order of the identified system, n to 10. Because of the long time a single experiment takes, only one trial was calculated here. The results are shown in table 2. Also here, the reported VAF is the average over all 12 outputs.

Next we used the 15 dataset from [6] that were taken of DAISY. On these datasets, we identified a model, with all six methods. The block matrix parameter i was set to the maximum value used in [6]. The order of the identified model was set to half this value. For all algorithms, we measured the number of FLOPS and the amount of CPU time in seconds. Because the order is chosen "randomly" , the resulting models are not of particular quality, and the VAF is not shown, (i.e. very bad in most cases). The main point of this test is to show the different performances of the algorithms on real-life sized datasets. For every dataset, we estimated the model 10 times. The CPU-time we report is the average over these ten trial. The dimensions of the datasets are summarised in table 3.

In table 4 and 5 the resulting number of FLOPS and CPU time is given. Table 4 shows that CVA has significantly less FLOPS than MOESP and N4SID. This is in correspondence with the results from [6]. MOESP and N4SID are about equal in this sense.

In table 5 we see a different picture. Of the three original algorithms, CVA is clearly the slowest. Although the number of floating point operations is smaller than for MOESP and N4SID, the computation time is worse. This can be explained by the fact that the CVA method uses a constrained least squares method to solve for A and C. This requires the construction of a large matrix, containing zeros and ones. This is not not counted as FLOPS but take significant time nonetheless. Most likely, the CVA algorithm can be optimised significantly. In that case we expect the algorithms to be more equal in respect of computation time. Figure 5 shows the number of FLOPS and the CPU time in a graphical way. This gives a clear indication of the relative speed and operations of the algorithms.

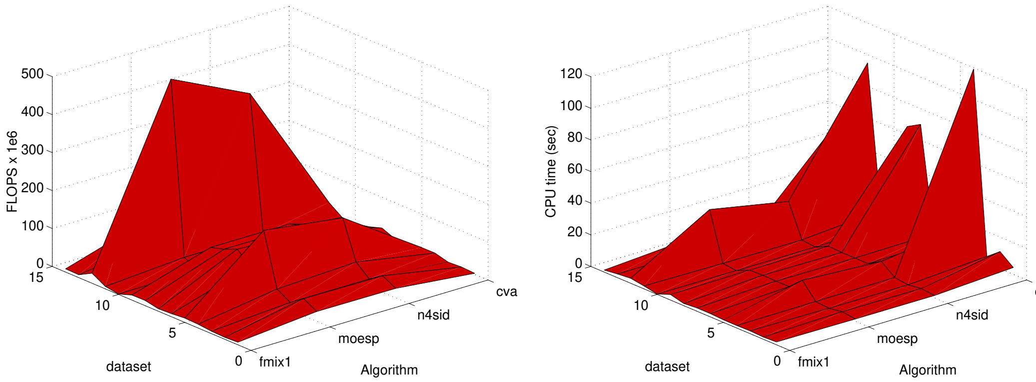

Figure 1: Measured flopcount(left) and CPU time (right) of the tested algorithms, on the DAISY datasets.

	mix1	fmix1	fmoesp	moesp	n4sid	cva
Mflops	8.6219	1.2773	2.7734	10.1970	9.3933	1.2183
CPU time	0.5400	0.4860	0.6450	0.7080	0.6480	0.7380
VAF	99.7870	99.7870	99.8818	99.8818	99.7071	99.2745

Table 1: Resulting million floating point operations (MFLOPS), computation time and variance accounted for (VAF) for model 1, for all six algorithms

	mix1	fmix1	fmoesp	moesp	n4sid	cva
Mflops	4287.6	580.9	1768.7	5484.0	5130.7	911.1
CPU time	198	29	165	341	236	59791
VAF	99.9837	99.9837	99.9844	99.9844	99.9841	99.9841

Table 2: Resulting million floating point operations (MFLOPS), computation time and variance accounted for (VAF) for model 2, for all six algorithms

dataset	m	l	n	i	N
ball & beam	1	1	12	24	1000
cd player arm	2	2	7	14	1500
dryer 1	1	1	12	24	750
dryer 2	3	3	7	14	600
evaporator	3	3	5	10	5000
heat exchanger	1	1	12	24	3000
flexible structure	2	1	7	14	6000
glass furnace	3	6	5	10	1000
pH data	2	1	7	14	1500
powerplant	5	3	2	4	200
robot arm	1	1	12	24	800
steam generator	4	4	5	10	7000
thermic wall	2	1	10	20	1000
winding process	5	2	5	10	1500

Table 3: The dimensions of the DAISY datasets used. m is the number of inputs, l the number of outputs, i the used block matrix dimension parameter, n the order of the identified model, and N the number of samples in the dataset.

dataset	**mix1**	**fmix1**	**fmoesp**	**moesp**	**n4sid**	**cva**
ball & beam	22.9173	6.7698	9.4587	25.8039	24.8189	2.7857
cd player arm	39.3372	5.0125	10.6355	45.2030	43.4158	4.7362
dryer 1	18.0562	6.3979	8.3028	20.0618	20.0312	2.6546
dryer 2	36.4539	9.9721	14.0574	40.8338	50.2641	12.0438
evaporator	146.7591	11.2746	44.0044	180.6030	151.8486	11.5370
heat exchanger	60.2864	8.2384	18.6799	71.1663	62.1980	3.7204
flexible structure	87.2851	6.4728	20.1506	101.8250	88.9316	4.6604
glass furnace	68.4128	13.5945	31.6528	86.8306	86.2001	18.9537
pH data	69.0229	13.0650	20.3254	76.7501	76.7783	7.7484
powerplant	2.0040	0.8732	1.5681	2.7210	2.7979	0.7698
robot arm	18.9900	6.3736	8.4655	21.1692	20.7985	2.5947
steam generator	364.0110	26.4220	137.1440	476.0641	375.9107	27.2481
thermic wall	32.7055	7.5004	10.4479	35.8716	37.3519	4.4906
winding process	61.2757	8.5658	18.6928	71.7759	69.3506	8.3406

Table 4: Millions of floating point operations (MFLOPS), over each method, for fixed value of i and n, for each dataset.

	mix1	fmix1	fmoesp	moesp	n4sid	cva
ball & beam	1.3890	1.7740	1.8280	1.4570	1.4180	4.3380
cd player arm	2.0950	1.3610	1.7950	2.5620	2.3480	10.8990
dryer 1	1.1520	1.7470	1.7550	1.1690	1.1280	4.2730
dryer 2	1.7040	1.5640	1.7210	1.8910	2.2700	119.1820
evaporator	7.8980	1.6090	5.3990	11.8260	8.8890	14.8940
heat exchanger	3.5570	2.0570	2.6390	4.1930	3.9300	4.4950
flexible structure	3.5570	2.0570	2.6390	4.1930	3.9300	4.4950
glass furnace	3.1690	1.3840	2.6180	4.4180	3.9190	70.1270
pH data	3.9060	3.5200	3.6590	4.1090	4.3820	65.2170
powerplant	0.1620	0.2010	0.2470	0.2170	0.2280	0.8290
robot arm	1.2460	1.8470	1.8440	1.2630	1.2470	5.0840
steam generator	19.7180	3.1220	15.0800	31.7880	22.0030	94.7830
thermic wall	1.9350	2.3040	2.3430	1.9930	2.1350	21.5990
winding process	3.1610	1.2430	2.7380	4.6930	3.7360	53.4470

Table 5: Average CPU time, over 10 trials for each method, for fixed value of i and n, for each dataset.

6 Conclusions

In this paper we have studied the computational complexity of three different SMI algorithms, namely MOESP, N4SID and CVA. It was shown that although CVA has a significantly lower number of floating point operations, the CPU time needed for the calculations is much higher than for N4SID and MOESP.

In order to improve the calculation time, we have developed a new algorithm based on a combination of MOESP and N4SID. This resulted in a significant increase in the calculation speed, compared with the original algorithms. Further increase in calculation speed was made by improving parts of the algorithm, that were most time consuming. Further improvements are possible, by improving the speed of the QR factorisation. This is a topic for further research.

References

[1] W. Larimore, "Canonical variate analysis in identification, filtering and adaptive control," in *Proceedings of the 29th IEEE Conf. Decision and Control*, (Hawaii), pp. 596–604, 1990.

[2] M. Verhaegen and P. Dewilde, "Subspace model identification part 1. The output-error state space model identification class of algorithms," *International Journal of Control*, vol. 56, no. 5, pp. 1187–1210, 1992.

[3] M. Moonen, B. de Moor, and J. Vandewalle, "SVD-based subspace methods for multivariable continuous-time systems identification," in *Identification of continuous-time systems* (N. Sinha and G. Rao, eds.), pp. 473–488, Kluwer Academic Publishers, 1991.

[4] M. Verhaegen, "Accurate identification of the temperature-product quality relationship in a multi-component distillation column," *To appear in Chemical Engineering Communications*, January 1997. Invited Paper.

[5] B. Haverkamp, "Identification of the human ankle dynamics using state space methods. a95041(703)," Master's thesis, Delft University of Technology, September 1995.

[6] W. Favoreel, S. van Huffel, B. de Moor, V. Sima, and M. Verhaegen, "Comparative study between three subspace identification algorithms." niconet, August 1998.

[7] N. Ramsey, "The spidery WEB system of structured documentation.".

[8] M. Potse, "MWEB user manual." Department of Medical Physics, University of Amsterdam, 1997.

[9] M. Potse, "The webfiles package." Department of Medical Physics, University of Amsterdam, 1997.

[10] D. T. Westwick, R. E. Kerney, and M. Verhaegen, "An efficient implementation of the PI subspace state-space system identification algorithm," in *IEEE conference on Engeering, Medicine and Biology*, vol. 18, 1997.

[11] J. Chun, T. Kailath, and H. Lev-ari, "Fast parallel algorithms for qr and triangular factorization," *SIAM Journal on Scientific and Statistical Computing*, vol. 8, no. 6, pp. 899–913, 1987.

[12] T. Kailath and J. Chun, "Generalized displacement structure for block-toeplitz, toeplitz-block and toeplitz-derived matrices," *SIAM Joural on Matrix Analysis and Applications*, vol. 15, no. 1, pp. 114–128, 1994.