Model Order Reduction of Large Scale ODE Systems:
MOR for ANSYS versus ROM Workbench

A.J. Vollebregt¹, T. Bechtold², A. Verhoeven³, E.J.W. ter Maten²,³

¹ Bergische Universität Wuppertal
² Philips Semiconductors - NXP, Eindhoven
³ Technical University of Eindhoven

Summary. In this paper we compare the numerical results obtained by different model order reduction software tools, in order to test their scalability for relevant problems of the microelectronic-industry. MOR for ANSYS [2] is implemented in C++ and ROM Workbench [3] is a MATLAB code. The chosen benchmarks are large scale linear ODE systems, which arise from the finite element discretisation of electro-thermal MEMS models.

1 Introduction

Decreasing size of silicon chips and their increasing integration density require permanently new and more powerful simulation tools and strategies in microelectronics and microsystem technology. Model order reduction (MOR) approaches [1] are successfully used to considerably reduce both the computational time and the resources. Mathematical development of MOR is an active area of research, which is growing from the reduction of linear ordinary differential equation systems (ODEs) towards the reduction of parameterized and nonlinear differential-algebraic equations (DAEs) and partial differential-algebraic equations (PDAEs). The implementation aspects of model order reduction are advancing as well. Practical MOR has developed from academic prototyping environments to several strong tools that can be easily used as an extension of the commercial simulators like e.g. ANSYS [2].

In today’s age of fast computers it is possible to use quick prototyping tools like MATLAB or Mathematica for convenient implementation and testing of new MOR methods. However, the run time for the usually large-scale industry relevant problems enforced the use of programming languages, like for example C++. Such implementations offer better performances, but also demand more time and programming skills from the developer. ROM Workbench [3] also provides a learning environment using MATLAB.

The goal of this paper is to numerically compare two MOR tools, which belong to the described streams: MOR for ANSYS (M4A) [2] and ROM Workbench (RW) [3]. The first was developed at the university of Freiburg, Germany, as an extension to the commercial finite element simulator ANSYS. However, it can be easily coupled to an arbitrary circuit simulator, provided the matrices of the linear dynamical system are exported in the Matrix Market [4] format. It implements block Arnoldi
algorithm from [5] and SOAR from [6]. RW is a MATLAB library of different MOR methods, which has been developed at the University Politehnica of Bucharest, Romania, within the European project CODESTAR. It implements a PRIMA version using block-Arnoldi based on [8]. Both tools are planned for use in the European project COMSON [9], which joins the efforts of the major European semiconductor companies and academic nodes to develop a demonstrator platform in a software code, that could fulfill the demands of the modern microelectronic industry. Such a comparison will give us a clear understanding up to which size and for what structure of the industrial problem the MATLAB code can be used and at which point one should switch to the compiled language implementation.

In section 2 we prove the equivalence of algorithms [5] and [8]. In section 3 we comment on the implementations within two codes and describe two electro-thermal MEMS (micro-electro-mechanical-systems) devices used as case studies for model order reduction. In section 4, the numerical results for order reduction with both tools are presented. Lastly, in section 5, we conclude the paper and describe some future directions.

2 Block Arnoldi Algorithms

In microsystem simulation, the spatial discretisation of computational domain often results in a linear multiple-input multiple-output ODE systems of the form

\[
C \cdot \dot{x} + G \cdot x = B \cdot u(t) \\
y = L^T \cdot x ,
\]

with initial condition \(x(0) = x_0\). Here, \(t\) is the time variable, \(x(t) \in \mathbb{R}^n\) the state vector, \(u(t) \in \mathbb{R}^m\) the input excitation vector and \(y(t) \in \mathbb{R}^p\) the output measurement vector. \(G, C \in \mathbb{R}^{n \times n}\) are linear (not depending on \(x\) and \(t\)) symmetric and sparse system matrices, \(B \in \mathbb{R}^{n \times m}\) and \(L \in \mathbb{R}^{n \times p}\) are (constant) input and output distribution arrays, respectively. \(n\) is the dimension of the system and \(m\) and \(p\) are the number of inputs and outputs.

Model order reduction is based on the projection of (1) onto some low-dimensional subspace. Most MOR methods generate two projection matrices \(V, W \in \mathbb{R}^{n \times n}\), to construct a reduced system of the order \(\nu\) as

\[
C_r \cdot \dot{z} + G_r \cdot z = B_r \cdot u(t) \\
y_r = L_r^T \cdot z ,
\]

with \(C_r = V^T CW, G_r = V^T GW, B_r = V^T B, \) and \(L_r = W^T L\). The ultimate goal of MOR is to find matrices \(V\) and \(W\) in such a way that \(\nu \ll n\), while minimizing the error between the full and the reduced system in either time domain \(\min \| y - y_r \|\) or Laplace domain. Furthermore, the stability and passivity of the original system should be preserved in 2.

The basic idea behind the Krylov-subspace based block-Arnoldi algorithm is to transfer (1) into the implicit (left-hand side) formulation

\[
A \hat{x} = x + Ru \\
y = L^T x ,
\]

with \(A = -G^{-1}C\), and \(R = -G^{-1}B\), and to write down the transfer function of (3) in the frequency domain, using a Taylor series in \(s_0\) as
\[ H(s) = -L^T (I - sA)^{-1} R = \sum_{i=0}^{\infty} m_i (s - s_0)^i, \]  

where \( m_i \equiv -L^T (I - sA)^{-1} B \) is called the \( i \)-th moment around \( s_0 \). One aims to find a reduced system whose transfer function \( H_r(s) \) will have the same moments as \( H(s) \) up to a degree \( \nu \). However, due to numerical instabilities, the moments are not computed explicitly, but via right-sided Krylov subspace \( K_r(A, R, \rho) := \text{span}(R, AR, A^2 R, \ldots, A^{\rho-1} R). \) Block Arnoldi algorithm generates a single orthonormal basis \( W \) for \( K_r(A, R, \rho) \) and the system (3) is reduced by projection to

\[ A_r z = z + R, u \\
 y_r = L_r^T z, \]  

with \( A_r = W^T AW, \ R_r = W^T R \) and \( L_r = W^T L \). The order of (5) is \( \nu = \rho \cdot m \). The property of the Krylov subspace is such that the first \( \nu \) moments of \( H_r(s) \) are identical to \( H(s) \) and \( H_r(s) \) are indeed identical.

As the reduced system (5) is not necessarily passive (this means that the system generates no energy, which property is important for applications in circuit simulation), two alternatives to "classical" block Arnoldi have been suggested: PRIMA algorithm [8] and Freund's Arnoldi [5]. Both are described and compared below.

### 2.1 PRIMA

The PRIMA algorithm was designed in 1998 to guarantee the passivity of the reduced system. PRIMA [8] stands for Passive Reduced-order Interconnect Macro-modeling Algorithm. Again, an orthonormal basis, \( X \), is generated such that \( \text{span}(X) = K_r(A, R, \rho) \), but \( X \) is used for an explicit projection of (1), such that \( C_r = X^T CX, \ G_r = X^T GX, \ B_r = X^T B \) and \( L_r = X^T L \). In [8] it is proven that for this reduced system the passivity is preserved if \( C \) is positively semi-definite and that the first \( n \) moments of the transfer function of the original and the reduced system are matched. Introducing the notation \( X_k = [x_{km+1} \ldots x_{(k+1)m}] \), an implementation of PRIMA can be found in Algorithm 1.

### 2.2 Freund's Arnoldi

Freund suggests in [5] that vectors which are almost linearly dependent with other vectors in the span of the orthonormal matrix should be eliminated. He calls this method of eliminating vectors deflation. His algorithm is vector based, although there is a block structure visible for multiple input multiple output systems. Instead of generating orthonormal blocks, the algorithm generates candidate vectors, and each vector \( \tilde{v}_k \) that satisfies

\[ ||\tilde{v}_k|| < DTOL, \]  

for some appropriate threshold \( DTOL \), is removed. Therefore, the number of vectors per orthonormal block \( m \) is smaller then, or equal to, the number of vectors of block \( m - 1 \). If this deflation is omitted we get Algorithm 2.
Algorithm 1 Block-Arnoldi as in PRIMA [8]
1: \( \hat{X}_0 = R \)
2: for \( j = 1, \ldots, m \) do
3: \( x_j = \hat{x}_j/\|\hat{x}_j\| \)
4: for \( i = j + 1, \ldots, m \) do
5: \( \hat{x}_i = \hat{x}_i - x_j x_j^T \hat{x}_i \)
6: end for
7: end for
8: for \( k = 1, 2, \ldots, \rho - 1 \) do
9: Determine \( \hat{X}_k = A\hat{X}_{k-1} \)
10: for \( j = 1, \ldots, k \) do
11: \( \hat{X}_k = \hat{X}_k - X_{k-j}\hat{X}_{k-j}^T \hat{X}_k \)
12: end for
13: for \( j = km + 1, \ldots, (k+1)m \) do
14: \( x_j = \hat{x}_j/\|\hat{x}_j\| \)
15: for \( i = j + 1, \ldots, (k+1)m \) do
16: \( x_j = x_i - x_j x_j^T x_i \)
17: end for
18: end for
19: end for

Algorithm 2 Freund’s Arnoldi ignoring deflation
1: for \( i = 1, \ldots, m \) do
2: \( \hat{v}_i = r_i, r_i \) being the \( i \)-th column of \( R \)
3: end for
4: for \( k = 1, 2, \ldots, \nu \) do
5: \( \hat{v}_k = \hat{v}_k/\|\hat{v}_k\| \)
6: Determine \( \hat{v}_{k+m} = Av_k \)
7: for \( i = 1, \ldots, k \) do
8: \( \hat{v}_{k+m} = \hat{v}_{k+m} - \hat{v}_i \hat{v}_i^T \hat{v}_{k+m} \)
9: end for
10: for \( i = k - 1, k - 2, \ldots, k - m + 2, k - m + 1 \) do
11: \( \hat{v}_{k+m} = \hat{v}_{k+m} - v_i \hat{v}_i^T \hat{v}_{k+m} \)
12: end for
13: end for

2.3 Comparison between both Algorithms

We suppose that the exact results of Algorithm 1 and Algorithm 2 are the same. In other words, Freund’s Arnoldi is PRIMA with deflation. We are now going to prove that this is indeed the case.

Proving that the exact results of both algorithms are the same, is equivalent to proving that \( X = V \) and that for \( V \) holds that

\[
\forall j, 0 < j \leq \nu: \quad v_j = \frac{\tilde{v}_j}{\|\tilde{v}_j\|}, \quad \tilde{v}_j = (1 - \sum_{i=1}^{j-1} v_i v_i^T) g(v_{j-m}), \tag{7}
\]

where \( g(v_{j-m}) = Av_{j-m} \) if \( j > m \) and \( g(v_{j-m}) = r_j \) if \( 0 < j \leq m \), \( g : \mathbb{R}^m \to \mathbb{R}^m \).

This is the wanted result of the original Arnoldi algorithm for multiple starting
vectors. So from this point on we will use $V$ as projection matrix for both algorithms. Introducing the notation
\[ q(v_j, \tau, \varphi) = (1 - \sum_{i=\tau}^{\varphi} v_i v_i^T) g(v_j), \]  

we need to prove that for all $j$ between 0 and $q$ we have $v_j = \frac{a(v_{j-1}, j-1)}{\|a(v_{j-1}, j-1)\|}$. Also, introduce the set of invariants

- $\Gamma(p) \equiv \{ v_j, 0 < j < p - 1 : v_j = \frac{a(v_{j-1}, j-1)}{\|a(v_{j-1}, j-1)\|} \}, \Gamma : \mathbb{N} \to \mathbb{B} = \{false, true\}$.
- $\Lambda(p, \tau, \varphi) \equiv \{ q(v_{\tau-1}, \tau, \varphi - 1) \}, \Lambda : \mathbb{N}^3 \to \mathbb{B}$.
- $\Omega(p, \tau, \varphi, \omega) \equiv \Lambda(p, \tau, \varphi) \land \ldots \land \Lambda(p + \omega - 1, \tau, \varphi), \Omega : \mathbb{N}^4 \to \mathbb{B}$.

If we can prove that $\Gamma(n+1)$ holds at the end of Algorithm 1 and Algorithm 2 we can conclude that $V = X$. We use Lemma 1 in both proofs.

**Lemma 1.** If for the set parameters $(\alpha, \beta, \gamma)$ the invariant $\Lambda(\alpha, \beta, \gamma)$ holds, then after
\[ v_\alpha = v_\alpha - v_j v_j^T v_\alpha, \]  

is executed, $\Lambda(\alpha, \beta, \gamma + 1)$ holds.

**Proof.** Assume $\Lambda(\alpha, \beta, \gamma)$ holds. Then
\[ v_\alpha = q(v_{\alpha-\beta}, \beta, \gamma - 1). \]  

Substitute this into (9) we get
\[ v_\alpha := (1 - v_j v_j^T)q(v_{\alpha-\beta}, \beta, \gamma - 1) \]
\[ = (1 - \sum_{j=\beta}^{\gamma-1} v_j v_j^T)g(v_{\alpha-\beta}) - v_j v_j^T [1 - \sum_{j=\beta}^{\gamma-1} v_j v_j^T]g(v_{\alpha-\beta})] \]
\[ = (1 - \sum_{j=\beta}^{\gamma} v_j v_j^T)g(v_{\alpha-\beta}) \]  

since $v_j^T v_j = 0$ for all $i$ and $j$, $i \neq j$.

Now we will prove that $\Gamma(n+1)$ holds at the end of both algorithms. We start with Algorithm 2 since it is less complicated.

**Theorem 1.** $\Gamma(n+1)$ holds at the end of Algorithm 2.

**Proof.** $\Gamma(1) \land \Omega(1, 1, m)$ holds after line $\beta$ in Algorithm 2, hence we are left with proving that $\Gamma(k) \land \Omega(k, 1, m)$ implies $\Gamma(k+1) \land \Omega(k+1, 1, k+1, m)$ after the loop that starts in line $4$. So assume we have we have $\Gamma(k) \land \Omega(k, 1, m)$ after line $4$. In line $5$ the current vector is normalized, and since $\Lambda(k, 1, k)$ holds we see that after this line we have $\Gamma(k+1) \land \Omega(k, 1, k)$. In line $6$ a new candidate vector is created, this adds $\Lambda(k + m, 1, 1)$. This implies that after line $6$ we have $\Gamma(k+1) \land \Omega(k, 1, k) \land \Lambda(k + m, 1, 1)$. Applying Lemma 1 we find that after line $9$ we have $\Gamma(k+1) \land \Omega(k, 1, k) \land \Lambda(k + m, 1, k+1)$. This is equivalent with $\Gamma(k+1) \land \Lambda(k, 1, k) \land \ldots \land \Lambda(k + m - 1, 1, k) \land \Lambda(k + m, 1, k+1)$. Again, using lemma
We find that for \( j = 1 \ldots m - 1 \) we can replace \( A(k + j, 1, k) \) with \( A(k + j, 1, k + 1) \). This gives
\[
\Gamma(k + 1) \land A(k, 1, k, k) \land A(k + 1, 1, k + 1) \land \ldots \land A(k + m, 1, k + 1) \\
\Rightarrow \Gamma(k + 1) \land \Omega(k + 1, 1, k + 1, m).
\]
This concludes our proof.

Theorem 2. \( \Gamma(\nu + 1) \) holds at the end of Algorithm 1.

Proof. Again, we start with \( \Gamma(1) \) and after the initialization of line 1 we have \( \Gamma(1) \land \Omega(1, 1, 1, m) \). Using Lemma 1 like in Theorem 1 we find after line 7 that we have \( \Gamma(m + 1) \). Now assume we have \( \Gamma(km + 1) \) after line 8. This is valid for \( k = 1 \), so for induction arguments we should find \( \Gamma((k + 1)m + 1) \) after line 18. In line 9, new candidate vectors are created. So for \( i \) equal \( km + 1 \) to \( (k + 1)m \) we get \( \mathbf{x}_i = \mathbf{G}(x_{i-m}) \). This gives \( \Gamma(km + 1) \land \Omega(km + 1, 1, 1, m) \equiv \Gamma(km + 1) \land \Omega(km + 1, 1, 1, m, 1) \). Now assume after line 10 we have \( \Gamma(km + 1) \land \Omega(km + 1, (k - j + 1)m + 1, km + 1, m) \). Apply \( \tilde{X}_k = \tilde{X}_k - X_{k-m} X_{k-m}^T \tilde{X}_k \). In vector terms this means that
\[
| \mathbf{x}_{km+1} | \ldots | \mathbf{x}_{(k+1)m} | = | \mathbf{x}_{km+1} | \ldots | \mathbf{x}_{(k+1)m} | - | \mathbf{x}_{(k-j)m+1} \mathbf{x}_{(k-j)m+1}^T \mathbf{x}_{km+1} | \ldots \mathbf{x}_{(k-j)m+1} \mathbf{x}_{(k+1)m} - \ldots - \mathbf{x}_{(k-j)m+1} \mathbf{x}_{(k-j)m+1}^T \mathbf{x}_{(k+1)m} = | \mathbf{A} \mathbf{x}_{(k-1)m+1} | - \sum_{i=(k-j)m+1}^{km} \mathbf{x}_i \mathbf{x}_i^T \mathbf{A} \mathbf{x}_{(k-1)m+1} | \ldots
\]
\[| A \mathbf{x}_{km} - \sum_{i=(k-j)m+1}^{km} \mathbf{x}_i \mathbf{x}_i^T A \mathbf{x}_{km} \Rightarrow \Omega(km+1, (k-j)m+1, km+1, m) \] if \( j \leq k \). This is the case in line 11. Therefore, after line 11 we have \( \Gamma(km + 1) \land \Omega(km + 1, (k-j)m+1, km+1, m) \), and the result of this loop will be \( \Gamma(km + 1) \land \Omega(km+1, 1, km+1, m) \). Again, a modified Gramm-Schmidt step is done for the lines 13 until 18. As we saw before, the result after line 18 is \( \Gamma((k+1)m + 1) \), which was the result we needed for our assumption to be correct. We may now substitute the \( \rho - 1 \) for the final result to get \( \Gamma(\rho m + 1) \). With \( \rho m + 1 = \nu + 1 \) we may conclude the proof.

3 Implementation and Case Studies

In the previous section we have proved the mathematical equivalence of the generated subspaces, while neglecting the numerical errors. In this section we will comment on the implementation of both algorithms within the software tools MOR for ANSYS and RW and will point out what adjustments we have made to the RW function in order to improve the performance for chosen case-studies.

3.1 MOR for ANSYS

MOR for ANSYS is an extension to the commercial finite element simulator ANSYS. It takes as input a linear ANSYS model (file full), reduces it and gives as output the matrices of the reduced system (2) in MatrixMarket format. However, for the purpose of the COMSON project it has been adjusted to also take as input the matrices of the arbitrary linear dynamical system (1). The code is a C++ implementation of Algorithm 2. The solve step in line 6 can be done with several forward-backward substitution methods (like LU- and Cholesky decomposition), which are available via the TAUCS-library. The reordering is done with METIS [7].
3.2 Rom Workbench (RW)

Rom Workbench is written in MATLAB. It implements several MOR methods including the Algorithm 1. The PRIMA function takes as input the matrices of the linear system (1) in MATLAB’s sparse array format. Unfortunately, the efficiency of this implementation is limited because it has no special treatment for symmetric $C$ and $G$, as the only available factorization is the LU decomposition of MATLAB with colamd as re-ordering scheme. However, the dynamical systems which arise from technical applications, as MEMS or electrical circuits, usually do have symmetric system matrices. As the COMSON Demonstrator Platform [9] should be able to handle a wide variation of industry-relevant problems, we have adjusted the PRIMA function of RW in such a way, that for symmetric matrices the performance is increased. We have implemented Cholesky-decomposition (as Cholesky is at least two times faster than LU decomposition) with symamd as re-ordering scheme. In the following we will call our adjusted version symRW.

3.3 Performance Phases

To analyze the bottlenecks of different implementations we divide the algorithm courses into several phases. These are

- Phase 1: Reading the original matrices into the memory from file and writing the reduced matrices to file.
- Phase 2: Reordering the matrix $G$.
- Phase 3: Factoring $G$ and constructing the first basis vector.
- Phase 4: Constructing the rest basis vectors via the back substitution in each iteration.

3.4 Case Studies

In order to test the presented MOR tools on industry-relevant problems, we have chosen two electro-thermal MEMS devices [10]. The pyrotechnical microthruster is based on the integration of the solid fuel with a silicon micro-machined structure. The thermally tunable optical filter is a Fabry-Perot interferometer fabricated as a free-standing membrane. Both models have been made and meshed in ANSYS (using low and high-order elements. In Table 1 the dimension of each test model and the matrix structure of the matrix $G$ are shown.

4 Numerical Results

We have reduced the described case studies using MOR for ANSYS, RW and it’s adjusted version symRW. In Fig. 1 a good match between the step response of the full-scale and that of the reduced order model at a single output node of the pyrotechnical microthruster are displayed. The difference between the reduced model computed with MOR for ANSYS and RW/symRW is of the order of rounding errors, as expected. In Tables 2 and 3 we compare the reduction time (down to order 30) of MOR for ANSYS, RW and symRW. CPU time of RW is up to 60 times longer than the CPU time of MOR for ANSYS. Due to our improvement, this difference has been reduced to 12 times for the largest case study. As expected the main speed up was achieved by introducing Cholesky factorization for the symmetric $G$ and a more effective ordering. The remaining CPU time difference is mainly due to the interpretation overhead in MATLAB.
### Table 1. Characteristics of MEMS benchmark

<table>
<thead>
<tr>
<th>Model</th>
<th>Dimension</th>
<th>Structure of $G$ (beginning part)</th>
<th>Model</th>
<th>Dimension</th>
<th>Structure of $G$ (beginning part)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1668</td>
<td>Tunable optical filter (low order elements)</td>
<td>B</td>
<td>106437</td>
<td>Tunable optical filter (high order elements)</td>
</tr>
<tr>
<td>C</td>
<td>26360</td>
<td>Pyrotechnical microthruster (low order elements)</td>
<td>D</td>
<td>79171</td>
<td>Pyrotechnical microthruster (high order elements)</td>
</tr>
</tbody>
</table>

**Fig. 1.** Step response of the full scale and reduced models (computed with MOR for ANSYS and RW) in a single output node of microthruster.

### 5 Conclusion

We have compared two software tools, which are meant to be integrated into the COMSON demonstrator platform. They belong to the two main implementation streams, fast prototyping in the interpreter environment and the compiled language implementation in C++. We have proven that both algorithms generate the same reduced basis and that the most important bottleneck for MATLAB is the decomposition phase. We have implemented Cholesky factorization for the symmetric problems in RW and have switched to a symamd re-ordering. Hence, the present run times in MATLAB allow for testing moderate-size industry-relevant problems within the COMSON demonstrator platform.
Table 2. Computational times in seconds on AMD Opteron with 2.4 GHz and 16 Gb RAM.

<table>
<thead>
<tr>
<th>Model A</th>
<th>M4A</th>
<th>RW</th>
<th>symRW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phase 1</td>
<td>5.9s-1</td>
<td>1.0s-1</td>
<td>6.0s-1</td>
</tr>
<tr>
<td>Phase 2</td>
<td>4.1s-2</td>
<td>1.0s-1</td>
<td>4.0s-1</td>
</tr>
<tr>
<td>Phase 3</td>
<td>5.6s-2</td>
<td>2.0s-2</td>
<td>5.0s-1</td>
</tr>
<tr>
<td>Phase 4</td>
<td>6.9s-2</td>
<td>6.0s-2</td>
<td>1.0s-1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Model B</th>
<th>M4A</th>
<th>RW</th>
<th>symRW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phase 1</td>
<td>7.2s-1</td>
<td>1.0s-1</td>
<td>7.0s-1</td>
</tr>
<tr>
<td>Phase 2</td>
<td>3.2s-2</td>
<td>2.0s-1</td>
<td>3.0s-0</td>
</tr>
<tr>
<td>Phase 3</td>
<td>3.9s-2</td>
<td>4.0s-2</td>
<td>3.9s-0</td>
</tr>
<tr>
<td>Phase 4</td>
<td>9.2s-2</td>
<td>7.0s-2</td>
<td>3.9s-1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Model C</th>
<th>M4A</th>
<th>RW</th>
<th>symRW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phase 1</td>
<td>2.3s-0</td>
<td>1.0s-1</td>
<td>2.0s-0</td>
</tr>
<tr>
<td>Phase 2</td>
<td>1.6s-1</td>
<td>1.3s-0</td>
<td>2.0s-0</td>
</tr>
<tr>
<td>Phase 3</td>
<td>2.5s-0</td>
<td>2.0s-2</td>
<td>4.0s-1</td>
</tr>
<tr>
<td>Phase 4</td>
<td>2.0s-0</td>
<td>1.1s-0</td>
<td>7.0s-0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Model D</th>
<th>M4A</th>
<th>RW</th>
<th>symRW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phase 1</td>
<td>1.1s-1</td>
<td>1.0s-1</td>
<td>1.0s-1</td>
</tr>
<tr>
<td>Phase 2</td>
<td>4.1s-0</td>
<td>7.0s-0</td>
<td>2.0s-0</td>
</tr>
<tr>
<td>Phase 3</td>
<td>4.1s-0</td>
<td>6.0s-0</td>
<td>3.0s-0</td>
</tr>
<tr>
<td>Phase 4</td>
<td>1.7s-0</td>
<td>1.1s-0</td>
<td>7.0s-0</td>
</tr>
</tbody>
</table>

Table 3. Complete reduction times in s for all the case studies on AMD Opteron with 2.4 GHz and 16 Gb RAM. \( n_{nz} \) is the number of nonzero matrix elements of \( \mathbf{G} \) and its factor \( \mathbf{L} \).

<table>
<thead>
<tr>
<th>Model</th>
<th>( n )</th>
<th>( n_{nz}(\mathbf{G}) )</th>
<th>M4A time</th>
<th>( n_{nz}(\mathbf{L}) )</th>
<th>RW time</th>
<th>( n_{nz}(\mathbf{L}) )</th>
<th>symRW time</th>
<th>( n_{nz}(\mathbf{L}) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1668</td>
<td>6.21e3</td>
<td>1.91e-1</td>
<td>2.46e-1</td>
<td>2.00e-1</td>
<td>3.39e-4</td>
<td>2.00e-1</td>
<td>2.32e-4</td>
</tr>
<tr>
<td>B</td>
<td>106437</td>
<td>1.41e6</td>
<td>4.55e1</td>
<td>1.89e7</td>
<td>4.02e-1</td>
<td>4.82e7</td>
<td>1.14e2</td>
<td>2.84e7</td>
</tr>
<tr>
<td>C</td>
<td>26360</td>
<td>2.65e5</td>
<td>7.34e0</td>
<td>5.00e6</td>
<td>2.16e2</td>
<td>1.68e7</td>
<td>5.38e1</td>
<td>1.06e7</td>
</tr>
<tr>
<td>D</td>
<td>79171</td>
<td>2.22e6</td>
<td>9.85e1</td>
<td>4.56e7</td>
<td>6.60e3</td>
<td>1.88e8</td>
<td>1.66e3</td>
<td>1.24e8</td>
</tr>
</tbody>
</table>

Acknowledgment

We would like to thank Dr. Evgenii B. Rudnyi from the University of Freiburg for helping us with MOR for ANSYS, Prof. Dr. Gabriela Ciuprina from the University Politehnica of Bucharest for her help with ROM Workbench and to acknowledge the EU support through the COMSON project.

References

2. www.intek.de/simulation/moransys
3. www.mek.ubp.de/codestar


9. www.comson.org

10. Oberwolfach Model Reduction Benchmark Collection, www.intek.de/simulation