ANALYTICAL MATRIX INVERSION AND CODE GENERATION FOR LABELING FLOW NETWORK PROBLEMS

Michael Weitzel and Wolfgang Wiechert
Institute of Systems Engineering / Simulation Group
Mechanical / Electrical Engineering & Computer Science
University of Siegen, 57068 Siegen, Germany
E-mail: michael.weitzel@uni-siegen.de

KEYWORDS
Analytical Matrix Inversion, Sparse Systems of Linear Equations, Code Generation.

ABSTRACT

Symbolic simplification and algebraic differentiation are only some of the advantages symbolic computations offer over their numerical counterparts. With a focus on sparse systems of linear equations, this contribution presents an analytical approach to a matrix inversion problem occurring in the field of Metabolic Flux Analysis. It is illustrated how the inherent complexity of the approach can be handled and how symbolic solutions can be compiled into highly performant machine code. Finally, benchmark results demonstrate that compiled analytical solutions offer comparable or even better performance than state-of-the-art sparse matrix algorithms.

INTRODUCTION

The research context of this contribution is given by *Metabolic Flux Analysis* (MFA), a class of network flow problems found in Systems Biology. One especially successful approach to MFA is based on isotopic labeling where a specifically labeled substrate, e.g. ^{13}C labeled glucose, is fed to the cells of an (typically unicellular) organism. According to the intra-cellular reaction rates (*flux values*) the isotopic labeling distributes among the cell's metabolites (Wiechert 2001). In the following, only those details are introduced which are necessary to understand the underlying mathematical structures and the described algorithms.

The computational heart of MFA is the simulation of isotopic labeling enrichment by the solution of a cascaded system of linear equations (Wiechert and Wurzel 2001):

$$\begin{aligned} {}^0\mathbf{x} &= \mathbf{1} \\ \mathbf{0} &= {}^k\mathbf{A}(\mathbf{v})\,{}^k\mathbf{x} + {}^k\mathbf{b}\left(\mathbf{v}, {}^1\mathbf{x}, \ldots, {}^{k-1}\mathbf{x}, {}^k\mathbf{x}_{inp}\right) \quad (1) \\ &\quad \text{for } k = 1, 2, \ldots \end{aligned}$$

These systems are obtained by serializing cascaded flow network graphs into balance equations. The unknowns ${}^k\mathbf{x}$ represent the labeling state of metabolites; the so called *cumulative isotopomer fractions*. The reaction rates $\mathbf{v}$ connecting the metabolites as well as the labeled input substrate ${}^k\mathbf{x}_{inp}$ are variables of the system. Moreover, vectors ${}^1\mathbf{x}, \ldots, {}^{k-1}\mathbf{x}$ serve as additional input for the system on level k and go into vector ${}^k\mathbf{b}$ in a nonlinear way. The described scheme is illustrated by the tiny example network in Fig. 1. The number of cascade levels corresponds to the maximum number of atoms (*labeling positions*) in a metabolite. The two connected flow network graphs are serialized into a cascaded system of balance equations. More details can be found in (Weitzel, Wiechert, and Nöh 2007).

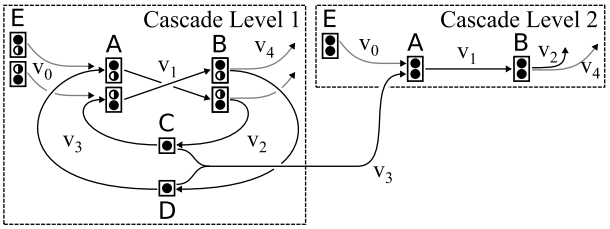

Figure 1: Cascaded flow network graphs

Cascade level 1:

$${}^1\mathbf{A}(\mathbf{v}) = \begin{pmatrix} -v_0 - v_3 & 0 & 0 & 0 & 0 & v_3 \\ 0 & -v_0 - v_3 & 0 & 0 & v_3 & 0 \\ 0 & v_1 & -v_1 & 0 & 0 & 0 \\ v_1 & 0 & 0 & -v_1 & 0 & 0 \\ 0 & 0 & v_2 & 0 & -v_2 & 0 \\ 0 & 0 & 0 & v_2 & 0 & -v_2 \end{pmatrix}$$

$${}^1\mathbf{x} = \begin{pmatrix} \mathrm{A}\bullet\bullet\bullet & \mathrm{A}\bullet\bullet\bullet & \mathrm{B}\bullet\bullet & \mathrm{B}\bullet\bullet & \mathrm{C}\bullet & \mathrm{D}\bullet \end{pmatrix}^T$$

$${}^1\mathbf{b}(\mathbf{v}, {}^1\mathbf{x}_{inp}) = \begin{pmatrix} v_0 \cdot (\mathrm{E}\bullet\bullet\bullet) & v_0 \cdot (\mathrm{E}\bullet\bullet\bullet) & 0 & 0 & 0 & 0 \end{pmatrix}^T$$

Cascade level 2:

$${}^2\mathbf{A}(\mathbf{v}) = \begin{pmatrix} -v_0 - v_3 & 0 \\ v_1 & -v_1 \end{pmatrix} \qquad {}^2\mathbf{x} = \begin{pmatrix} \mathrm{A}\bullet\bullet & \mathrm{B}\bullet\bullet \end{pmatrix}^T$$

$${}^2\mathbf{b}(\mathbf{v}, {}^1\mathbf{x}, {}^2\mathbf{x}_{inp}) = \begin{pmatrix} v_0 \cdot (\mathrm{E}\bullet\bullet) + v_3 \cdot (\mathrm{C}\bullet) \cdot (\mathrm{D}\bullet) \\ 0 \end{pmatrix}$$

In practice, each level of these cascaded *compartmental systems* typically contains balance equations for a few dozens to several thousands of unknowns and possesses certain structural properties (Weitzel, Wiechert, and Nöh 2007):

Proceedings of the 2008 High Performance
Computing & Simulation Conference ©ECMS
Waleed W. Smari (Ed.)
ISBN: 978-0-9553018-7-2 / ISBN: 978-0-9553018-6-5 (CD)

1. Each edge of the flow network graph labeled with a flux value v_i subtracts v_i from the diagonal of ${}^k\mathbf{A}$ and adds the same v_i to an off-diagonal element. This property results in *weak diagonal dominance* of ${}^k\mathbf{A}$ since the diagonal elements sum-up a node's total influx with negative sign.

2. Metabolic networks are usually sparse graphs (Weitzel et al. 2007). The ratio of the number of edges to the number of nodes (i.e. the *connectivity*) ranges from 2 to 2.5. This means that the number of edges is in the order of the number of nodes. This property results in sparse matrices ${}^k\mathbf{A}$.

3. It can be shown that the connectivity of graphs decreases monotonously with increasing cascade level, thus matrices ${}^k\mathbf{A}$ are increasingly sparse.

4. Except for the diagonal, matrices ${}^k\mathbf{A}$ have the same non-zero pattern as the transposed adjacency matrix of the flow network graph. This property is invariant with respect to flux values $\mathbf{v}$ as long as $v_i \neq 0$.

5. The flow network graphs associated with ${}^k\mathbf{A}$ typically contain a wide range of isomorphic subgraphs (Weitzel et al. 2007).

6. Only a few variables $\mathbf{v}$ determine matrices ${}^k\mathbf{A}(\mathbf{v})$. Moreover, with increasing k, the variety of flux variables is cut back.

7. By using a certain permutation $\mathbf{P}_k$, matrices ${}^k\mathbf{A}$ can be permuted into a upper (lower) block-triangular form $\mathbf{P}_k^T({}^k\mathbf{A})\mathbf{P}_k$ where the blocks on the diagonal are compartmental matrices, again.

Strategies for Solving the Cascaded System

The straightforward approach for the solution of Eq. (1) would be to choose an efficient implementation of a standard LU algorithm, such as LAPACK's `DGETRF` / `DGESV` (Anderson et al. 1999). These standard implementations use partial pivoting in order to provide numerical robustness. For a reasonably large network, where $k = 1, \ldots, 11$ and $\max_k\{\dim({}^k\mathbf{A})\} \approx 1200$, this results in about 20 sec. for a single simulation run. However, by taking advantage of the special structure of Eq. (1), the running time can be reduced to about $20 \cdot 10^{-3}$ sec. for the same system (Weitzel, Wiechert, and Nöh 2007).

Diagonal dominance (property 1) can be used to speedup the classical LU algorithm by skipping the partial pivoting phase completely without risking numerical stability (Golub and van Loan 1996).

Sparsity of matrices ${}^k\mathbf{A}$ (property 2.) suggests to use a specialized sparse matrix algorithm. One possibly senseful way to use this property is to solve the *bandwidth-reduced* system $(\mathbf{Q}^T\mathbf{A}\mathbf{Q})(\mathbf{Q}^T\mathbf{x}) + \mathbf{Q}^T\mathbf{b} = \mathbf{0}$ by using a banded-LU algorithm. Property 4 ensures that the bandwidth-reducing permutation $\mathbf{Q}$ can be computed in advance, since ${}^k\mathbf{A}(\mathbf{v})$ does not change its non-zero pattern if $\mathbf{v}$ is varied. Because the algorithm for determining $\mathbf{Q}$ is a heuristic (George and Liu 1981) and significant bandwidth reduction is not always possible this reduces the computation time for the above example network from 20 sec. to about four sec.

The basis for more successful algorithms is the upper block-triangular form $\mathbf{P}_k^T({}^k\mathbf{A})\mathbf{P}_k$. In case only the diagonal blocks contain non-zero values each block corresponds to an independent subsystem. These subsystems do not influence each other and can be solved in any order. In the more general case, where diagonal blocks are *connected* by coefficients above the block diagonal, the subsystems given by the diagonal blocks are unilaterally dependent and thus, have to be solved in a specific order during a single back-substitution run (Davis 2006). The described algorithm greatly benefits from decreasing connectivity (property 3) which leads to a large number of small subsystems and highly sparse matrices ${}^k\mathbf{A}$. The use of this algorithm results in the above mentioned speedup of about factor 1000.

By property 5, subgraph isomorphism is an important feature of graphs ${}^kG$ and thus, classes of *equivalent* subsystems (equivalent except for permutation) can be found in ${}^k\mathbf{A}$. Even though all subsystems of such an equivalence class share the same LU decomposition, the beneficial use of subgraph isomorphism is difficult in practice and involves costly management which usually foils the additional theoretic speedup at least for smaller subsystems. Since, on the other hand, larger subsystems are sparse as well it might seem beneficial to switch to a banded-LU in order to solve them. In practice, the obtained speedup is minimal: the number of large subsystems is small and due to their origin, the subsystems are ill-suited for bandwidth reduction.

Property 6 is of no use for all direct numerical methods. This changes for the analytical solutions presented in the next section.

ANALYTICAL SOLUTIONS

The Idea behind Analytical Solutions

Although there exist fast solution algorithms for positive definite, banded, sparse, or other special systems the running time is always governed by the structure of the solution algorithm, but at most roughly by the structure of the underlying problem (matrix). In case of systems where the coefficients in $\mathbf{A}$ may change their value, but the sparsity pattern remains the same, e.g. during a parameter variation study, common algorithms suffer from their static setup and specialized algorithms obtained from analytical solutions might do a better job.

Even though sparsity and isomorphism give the clue that additional speedup might be within reach, the classical direct numerical sparse matrix algorithms do not accomplish to capitalize these properties exhaustively. Furthermore, the presorting and graph analysis steps found in these algorithms do not care about *how* a certain non-zero

element is composed (i.e. the mathematical expression that lead to the non-zero value) since this information is usually out of reach and of no use for the algorithms found in conventional sparse matrix libraries.

The idea behind an analytical solution is simple. In general, analytical solutions can be obtained by carrying out an algorithm's operations symbolically: instead of executing arithmetic operators they are recorded symbolically in form of an expression tree. This works especially well if the underlying algorithm is branch-free, like the LU algorithm if partial pivoting is omitted.

Analytical solutions facilitate any type of *a posteriori* analysis. An interesting example is the ability to perform algebraic differentiation of Eq. (1) in order to provide highly accurate partial derivatives (*sensitivities*)

$$\frac{\partial \mathbf{x}(\mathbf{v})}{\partial \mathbf{v}} = -\frac{\partial}{\partial \mathbf{v}} \left({}^k\mathbf{A}(\mathbf{v})^{-1} \cdot {}^k\mathbf{b}(\mathbf{v}, \dots) \right)$$

used by various optimization algorithms during the parameter fitting step. Algebraic differentiation of explicit solutions is implemented as a set of simple expression tree transformation rules. Another example is the estimation of error propagation when a solution formula is actually evaluated using floating point arithmetic.

Analytical Matrix Inversion

Matrix inversion is usually not an option when solving a moderately sized system $\mathbf{A}\mathbf{x} + \mathbf{b} = \mathbf{0}$. Once a LU factorization of $\mathbf{A}$ is computed in $\Theta(n^3)$ FLOPS the solution of $\mathbf{A}\mathbf{x} + \mathbf{b} = \mathbf{0}$ requires additional $2n^2$ FLOPS (where $n = \dim \mathbf{x}$). In contrast, a matrix inversion based on a LU factorization requires $2n^3$ FLOPS, i.e. n times the effort of a solution. However, seen from an asymptotic standpoint, both direct numerical approaches require $\Theta(n^3)$ FLOPS (Golub and van Loan 1996).

Analytical computations require a change of perspective: there is no immediate relation between the time complexity of the algorithm *generating* the analytical solution and the number of FLOPS required to *evaluate* it; i.e. its size. While the underlying algorithm might not be able to handle sparsity efficiently, *structural* zeros in the input data cancel-out large parts of the analytical computations. In this context, it is helpful to adopt a classical idea from compiler design: high computational effort during the compilation phase and expensive optimizations are justified as long as the resulting code has improved efficiency and is executed frequently (Muchnick 1997).

For the original MFA problem, i.e. the solution of the cascaded system (1), the computation of an inverse $({}^k\mathbf{A})^{-1}$ becomes an option because only a small subset of the elements of $({}^k\mathbf{A})^{-1}$ is actually required to describe the MFA experiment:

- Measurements typically describe only a small subset of ${}^k\mathbf{x}$. A direct numerical approach for solving the cascade (1) cannot be trimmed to provide only the relevant subset of the solution vector ${}^k\mathbf{x}$. Clearly, having an analytical solution for $({}^k\mathbf{A})^{-1}$, this can be accomplished by extracting the relevant rows out of ${}^k\mathbf{x} = -({}^k\mathbf{A})^{-1}({}^k\mathbf{b})$.
- Vectors ${}^k\mathbf{b}$ are likewise sparse and a multiplication of type $({}^k\mathbf{A})^{-1}({}^k\mathbf{b})$ *selects* relatively few columns of $({}^k\mathbf{A})^{-1}$ for building the solutions ${}^k\mathbf{x}$.

Moreover, by the application of iterative improvement, an inverse $(\mathbf{A}(\mathbf{v}))^{-1}$ for flux vector $\mathbf{v}$ can be reused for some time as long as the parameter fitting procedure generates new flux vectors $\mathbf{w} = \mathbf{v} + \mathbf{d}$ by introducing slight changes $\mathbf{d}$:

$$\begin{aligned} \mathbf{x}_1 &\leftarrow -(\mathbf{A}(\mathbf{v}))^{-1}\mathbf{b}(\mathbf{w}) \\ \mathbf{x}_{i+1} &\leftarrow \mathbf{x}_i + (\mathbf{A}(\mathbf{v}))^{-1}(\mathbf{A}(\mathbf{w})\mathbf{x}_i - \mathbf{b}(\mathbf{w})). \end{aligned}$$

In practice the sequence $\mathbf{x}_1, \dots, \mathbf{x}_m \approx \mathbf{x}(\mathbf{w})$ converges to an acceptable result after few iterations. However, this is not applicable for larger $\mathbf{d}$. Furthermore, $m \ll n$ should hold, since each iteration requires $\Theta(n^2)$ FLOPS. See (Press et al. 2007) for details.

Other applications of the inverse include the implicit EULER iteration, where a non-stationary variant of Eq. (1) is considered.

Complexity of Analytical Solutions

Algebraically, the matrix inversion problem is closely related to the computation of the transitive, reflexive closure of a graph (Lehmann 1977). Each element of an inverse corresponds with a set of paths connecting a pair of nodes in the associated computational graph. Aiming to compute only a subset of ${}^k\mathbf{x}$ with the greatest possible efficiency this *path tracing* approach marked the starting point for our research. It was shown in (Isermann, Weitzel, and Wiechert 2004) how the approach can be applied to the solution of Eq. (1) using a generalized KLEENE algorithm. The resulting branch-free matrix inversion procedure is extremely simple and shall be used to illustrate the sketched scheme for building an analytical solution:

INVERSE $(\mathbf{E}_0 : \mathbb{R}^{n\times n}) : \mathbb{R}^{n\times n}$

1 $\mathbf{E} \leftarrow \mathbf{I}_n - \mathbf{E}_0$
2 **for** $k \leftarrow 1$ **to** n
3 **do** $\mathbf{E} \leftarrow \mathbf{E} + \dfrac{\mathbf{E}(:,k)\mathbf{E}(k,:)}{1 - \mathbf{E}(k,k)}$
4 **return** $\mathbf{E} \leftarrow \mathbf{I}_n + \mathbf{E}$

The following result can be proven by induction:

Proposition. *After executing k times line 3 the contents of $\mathbf{E}$ are*

$$\mathbf{E}_k = \begin{bmatrix} \mathbf{A}_k^{-1} - \mathbf{I}_k & -\mathbf{A}_k^{-1}\mathbf{B}_k \\ -\mathbf{C}_k\mathbf{A}_k^{-1} & \mathbf{I}_{n-k} - (\mathbf{D}_k - \mathbf{C}_k\mathbf{A}_k^{-1}\mathbf{B}_k) \end{bmatrix}$$

with the partition

$$\mathbf{E}_0 = \begin{array}{c} \\ k \\ n-k \end{array} \begin{array}{c} \begin{array}{cc} k & n-k \end{array} \\ \begin{pmatrix} \mathbf{A}_k & \mathbf{B}_k \\ \mathbf{C}_k & \mathbf{D}_k \end{pmatrix} \end{array}.$$

Moreover, after executing n times line 3, the contents of $\mathbf{E}$ are $\mathbf{A}_n^{-1} - \mathbf{I}_n = \mathbf{E}_0^{-1} - \mathbf{I}_n$. Hence, the inverse $\mathbf{E}_0^{-1}$ is returned in line 4.

Discarding the matrix notation, line 3 can be equivalently written as

3 $\mathbf{E}' \leftarrow \mathbf{E}$
3 **for** $i, j \leftarrow 1$ **to** n
3 **do** $\mathbf{E}(i,j) \leftarrow \mathbf{E}'(i,j)$
3 $+\mathbf{E}'(i,k)\mathbf{E}'(k,j)\dfrac{1}{1-\mathbf{E}'(k,k)}$.

The matrix operation in line 1, the three loops, and the addition of the identity matrix require $5n^3+n^2+n$ FLOPS in total, which is about two times slower than a matrix inversion based on the LU algorithm. Likewise, the required memory is in $\Theta(n^2)$. However, if *deferred evaluation* is used, i.e. the above code is used to obtain an expression tree for a symbolic rational expression, the memory requirements change dramatically.

The memory requirements $s(k)$ of a single entry of $\mathbf{E}_0$ are assumed to be in $\Theta(1)$. In step k of the outer loop the relaxation step in line 3 incorporates four times a value from the previous matrix $\mathbf{E}'$, introduces two $\Theta(1)$ nodes for the value "1" and another five $\Theta(1)$ nodes for operators. Hence, the recurrence $s(k) = \Theta(4s(k-1)+7)$ with $s(0) = \Theta(1)$ describes the memory consumption of the symbolic solution of a single element. The solution to this recurrence is

$$s(k) = \Theta\left(4^k + 7\sum_{i=0}^{k-1} 4^i\right) = \Theta((10\cdot 4^k - 7)/3). \quad (2)$$

For the complete matrix $\mathbf{E}_0^{-1}$ this results in expression trees having $\Theta(n^2 4^n)$ nodes in total (not counting the operations in lines 1 and 4). Clearly, this is intractable even for small problem sizes – for example, when assuming 12 bytes for a single expression tree node and $n = 10$ this results in 4000 megabytes for $\mathbf{E}_0^{-1}$.

Elimination of Common Subexpressions

At this point, analytical solutions for the matrix inversion problem seem to be completely impractical: even if the exponential memory requirements were no problem evaluation of the symbolic solution for each element of $\mathbf{E}_0^{-1}$ would require exponential time – just because the analytical solution has exponential size.

Fortunately, this problem can be handled by using the fact that the generated symbolic expressions share common subexpressions to great extend. A minimal example for $n = 2$ shall illustrate this. Just before the loop in line 2 starts the contents of $\mathbf{E}$ are

$$\mathbf{E} = \mathbf{I}_2 - \mathbf{E}_0 = \begin{pmatrix} a & b \\ c & d \end{pmatrix}.$$

When the loop of line 2 exits the contents of $\mathbf{E}$ are

$$\mathbf{E} = \mathbf{E}_0^{-1} - \mathbf{I}_2 = \begin{pmatrix} e & f \\ g & h \end{pmatrix}.$$

For instance, the formula representing element h is

$$h = d + \frac{cb}{1-a} + \left(d + \frac{cb}{1-a}\right)\left(d + \frac{cb}{1-a}\right)S$$

using the term $S := 1/(1-(d+cb/(1-a)))$ which is a common subexpression of all solutions.

All expression trees for the elements of $\mathbf{E}_0^{-1}$ have the same size, namely 51 nodes (26 leaves, ten nodes for multiplication and five divisions, additions, and subtractions), thus 204 nodes in total. This conforms with (2), since $2^2 s(2) = 4 \cdot 51 = 204$. Collapsing all common subexpressions of the expression trees in $\mathbf{E}_0^{-1} - \mathbf{I}_2$ transforms the *forest* into a single directed acyclic graph (DAG) with only 33 nodes (5 leaves, 16 multiplications, two divisions and subtractions, and eight additions), i.e. only 16% of the nodes found in the forest.

A Divide & Conquer Matrix Inversion Algorithm

Different matrix inversion algorithms result in completely different analytical solutions. Although there is no immediate relation between size of the analytical solution and the running time of the underlying algorithm there is at least a tendency that a less efficient algorithm results in larger analytical solutions.

Another branch-free matrix inversion algorithm having its roots in the path tracing formalism is based on a recursive formulation of the matrix inversion problem (Conway 1971, Zhang 2005, Press et al. 2007):

$$\begin{bmatrix} \mathbf{P} & \mathbf{Q} \\ \mathbf{R} & \mathbf{S} \end{bmatrix}^{-1} = \begin{bmatrix} \mathbf{T^{-1}} & \mathbf{-T^{-1}QS^{-1}} \\ \mathbf{-S^{-1}RT^{-1}} & \mathbf{S^{-1}(I+RT^{-1}QS^{-1})} \end{bmatrix}$$

with $\mathbf{T} \stackrel{\text{def}}{=} \mathbf{P} - \mathbf{QS^{-1}R}$ (the SCHUR complement of $\mathbf{S}$).

An efficient implementation of this formula results in a divide & conquer algorithm which requires two recursive calls for the inversion of $\mathbf{T}$ and $\mathbf{S}$ and inverts a matrix in about $2n^3 + 2n^2 - n/2$ FLOPS, which is slightly better than the LU algorithm without partial pivoting ($8n^3/3 - n^2/2 - n/6$).

More importantly, the DAGs containing the analytic solutions are about 20% smaller than those obtained from the KLEENE algorithm. Compared to the KLEENE algorithm an economical implementation of this algorithm is challenging and requires in-place matrix operations and careful handling of memory resources.

Creating DAGs and Handling Sparsity

Clearly, creating the analytical solutions in form of expression trees, first, and then collapsing them into DAGs in a second step would be an impractical strategy since this would require exponential memory. Instead, DAGs have to be created on-the-fly.

This is an especially elegant way because in each step of the matrix inversion algorithm a newly created operator node accesses only previously seen nodes. Using techniques like *common subexpression elimination*, *value*

numbering (Muchnick 1997), as well as algebraic properties, like commutativity of operators, the DAG generation algorithm is able to decide whether an operator node needs to be created, or an existing node can be reused.

Certain applications of operators may lead to the value zero. In most of these cases the operator is a multiplication and one of the operands is a zero. This situation can be handled by replacing the operator node by a reference to the value zero. The remaining non-zero operand is kept in memory since it may be used by a later computation. Finally, when all elements of the inverse have been computed, the DAG is pruned of the unused nodes. This strategy efficiently handles sparsity and identifies matrix elements which contain a *structural zero*, i.e. a zero value which originates in the structure of the actual inversion problem.

Machine Code Generation

Once the DAG representing the inverse $\mathbf{A}^{-1}$ is in memory the goal is to evaluate it as fast as possible. Since the DAG is acyclic this can be established using a simple depth-first search starting at the individual root nodes and stopping either at the leaf nodes, or at nodes that have been visited before. Although this approach is far better than evaluating the original forest, there is some overhead for dereferencing pointers, switching the different operators and managing node labels.

This overhead can be eliminated by compiling the DAG into byte-code – either for an interpreter, or into machine code for direct execution on a CPU. Another option would be to emit a programming language source code, e.g. C code, which is compiled and optimized using an existing compiler. The drawback of this option is that common compilers are designed for small expressions and small basic blocks and compilation may take too long or require too much memory.

In the ongoing project the DAGs are compiled into efficient machine-code for the FPUs found in x86 family CPUs. These FPUs are organized as stack machines.

Clearly, the DAG has to be evaluated in a topological order starting at its leaves and stopping at the DAG's roots, i.e. result nodes representing the elements of the inverse matrix. Following (Bruno and Lassagne 1975), this special topological order, together with some management instructions, can be described as an *abstract stack machine* program. By design, this *program* forces an evaluation if either the stack load exceeds the maximum capacity or a root of the DAG (i.e. a result node) is reached. Furthermore, the abstract stack machine program cares about reusing previously computed intermediate results and saving them for later use.

Based on the information contained in the DAG and the abstract stack machine program x86 machine code is generated. Since complicated issues, like the allocation of the stack registers, and the correct handling of common subexpressions are already treated during the assembly of the abstract stack machine program a single pass is sufficient for generating the x86 FPU code.

In a second pass, a *peephole optimization* performs local optimization by sliding over the generated code and removing suboptimal constructs within a certain window. Finally, some prolog and epilog are generated, the assembler code is compiled into an object file and linked into a dynamic library. This library is dynamically loaded once at runtime and the matrix inversion code can be accessed via a function pointer.

RESULTS

In this section the numerical accuracy of the presented KLEENE algorithm and the divide & conquer algorithm are compared in order to illustrate the appropriateness for the cascaded systems in Eq. (1). Next, the performance of the compiled analytical solutions is compared to `LAPACK` (Anderson et al. 1999) and `CSparse` (Davis 2006), a state-of-the-art sparse matrix library.

Empirical Comparison of Numerical Accuracy

Figures 2 and 3 give empirical results for the numerical stability of the different matrix inversion algorithms. Each curve shows the mean, the minimum, and maximum of the residual $||\widehat{\mathbf{A}^{-1}}\mathbf{A} - \mathbf{I}||_\infty$ of 100 randomly generated matrices $\mathbf{A}$ and the numerically computed inverses $\widehat{\mathbf{A}^{-1}}$. These results demonstrate the growth of numerical errors for fully populated and sparse matrices when doubling the matrix dimension and are based on direct numerical implementations, i.e. do not use analytical solutions. Figure 2 shows the results for matrices fully pop-

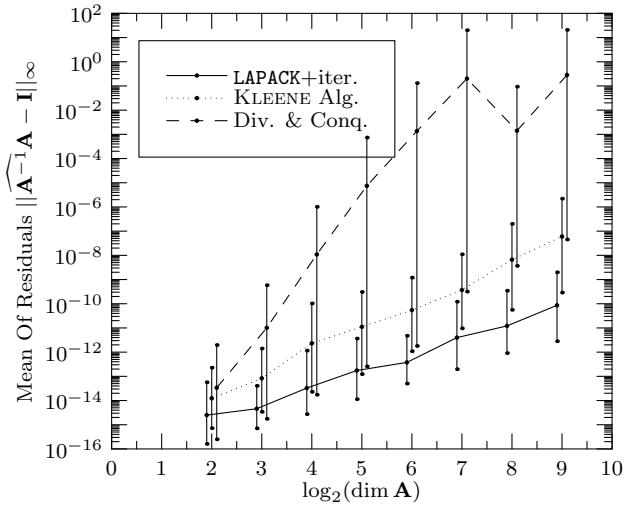

Figure 2: Numerical stability of different matrix inversion algorithms on fully populated random matrices. Shown are mean, min., and max. values of residuals.

ulated with $N(0, 1)$ random numbers. Matrices of this type are usually well-conditioned. The accuracy of the KLEENE algorithm is acceptable even for larger matrices. In contrast, the divide & conquer algorithm performs bad and the worst-case numerical accuracy is unacceptably

low for larger matrices. The solid line corresponds to a matrix inversion based on a combination of the LAPACK routines `DGETRF` / `DGETRI` and an additional iterative improvement and is given for reference.

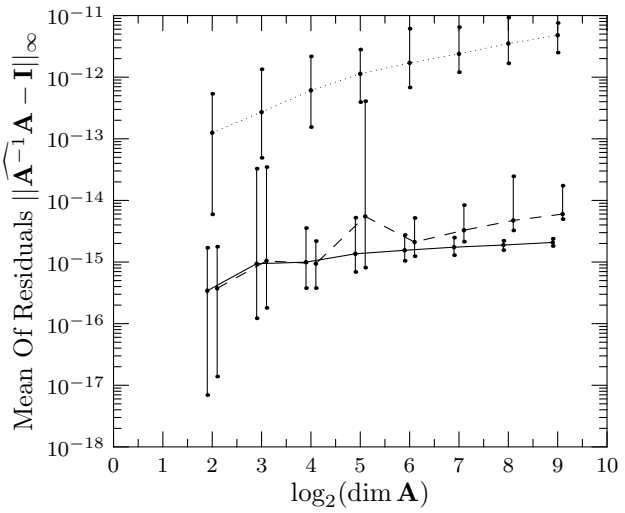

Figure 3: Numerical accuracy of the different matrix inversion algorithms on a set of 100 sparse random matrices obtained from randomly connected flow networks.

In Fig. 3 the same analysis was repeated for compartmental matrices obtained from randomly generated ERDÖS-REYNI flow network graphs (Bolobás 2001) assuming $\dim(\mathbf{A})$ in- and effluxes and $N(0,1)$ random flux values. The performance of all algorithms is very good. This time, amazingly, the divide & conquer algorithm returns the second best results.

Performance of Analytical Solutions

Figures 4, 5, and Fig. 6 compare the performance of the analytical solutions with LAPACK's `DGETRF` & `DGETRI` (Anderson et al. 1999) and the state-of-the-art sparse matrix library `CSparse` (Davis 2006). All benchmarks were prepared on a 2 GHz Pentium M machine with 2 GB of RAM. All analytical solutions are based on the presented divide & conquer algorithm.

Figure 4 shows the results for the set of fully populated random matrices. Because of the memory limitation during compile-time it was possible to compose analytical solutions for inverses of fully-populated matrices up to dimension 160.

Figure 5 gives the results for the set of random sparse matrices, again obtained from randomly connected flow networks. The maximum dimension achievable for sparse matrices ($\dim 2000$) depends on the degree of sparsity, which was adjusted to reflect a realistic network. The size of the machine code generated from sparse and fully populated matrices hitting the memory limit is comparable.

Finally, Fig. 6 shows the results for a realistic cascade obtained from a metabolic network representing the central metabolism of *E. Coli*. In contrast to the results presented in figures 4 and 5, the code generation algorithm worked on matrices ${}^k\mathbf{A}(\mathbf{v})$ containing symbolic expressions in its elements while `CSparse` worked on sparse matrices containing double precision floating point values.

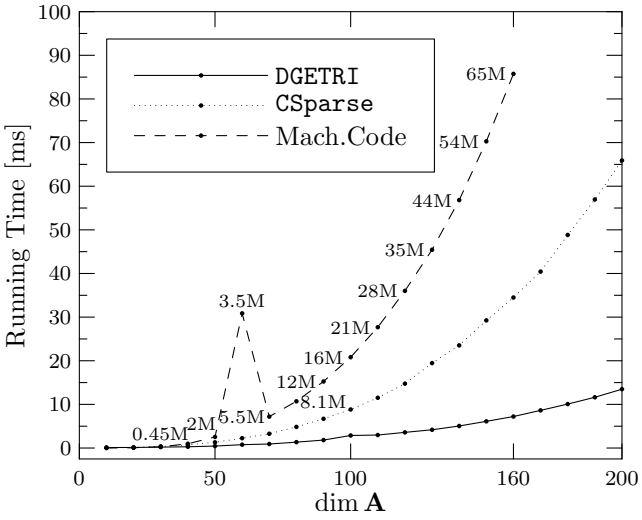

Figure 4: Running times of `LAPACK`, `CSparse`, and the generated machine code on fully populated matrices. Code sizes of the machine codes are given in megabytes.

CONCLUSIONS

Typical sparse matrix algorithms are based on the analysis of a system's non-zero pattern, or equivalently, the system's computational graph. The resulting techniques, like fill-in reducing permutations or system decomposition, are often based on elaborate graph-theoretical algorithms. Preconditioning and pivoting, on the other hand, analyze the matrix elements in order to provide numerical robustness. In contrast to the presented analytical matrix inversion, all these techniques may be characterized as *top-down* approaches.

Analytical solutions are constructed *bottom-up* using simple rules, which results in highly complex tailor-made solutions for a specific problem. Efficient handling of the inherent complexity and the choice of the underlying algorithm is crucial for the feasibility and the numerical stability of the approach. The divide & conquer algorithm qualifies in both aspects for the flow network problem.

Since the preparation of an analytical inverse may take from seconds to several minutes the effort has to be justified by the underlying problem (e.g. the use in a parameter variation study) and analytical computation is surely no option if an inverse is needed only once. While the generated code for the analytical inverse of fully populated matrices is slower than the LU algorithm used in LAPACK (Fig. 4), it is about three times faster than a state-of-the-art sparse matrix technique for randomly generated sparse matrices (Fig. 5) and up to five times faster for matrices obtained from a realistic network, where symbolic equations can be provided (Fig. 6).

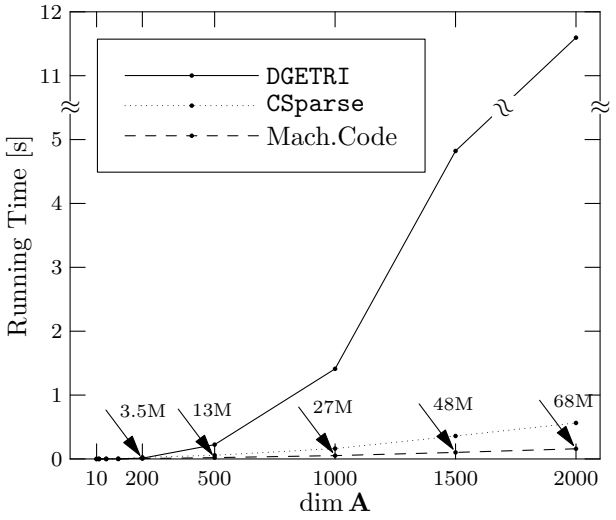

Figure 5: Running times of `LAPACK`, `CSparse` and the generated machine code on sparse matrices obtained from randomly connected flow networks with $N(0,1)$ randomly distributed flow values.

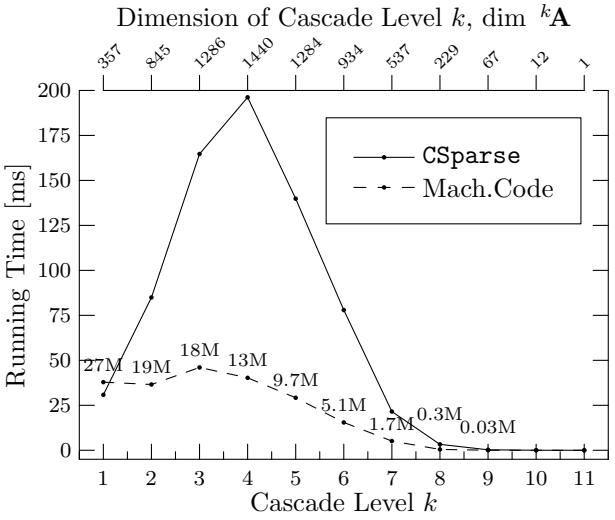

Figure 6: Running times on a realistic cascaded network representing the central metabolism of *E. Coli*. Sparsity increases monotonously with cascade level k and determines code size and running time.

FURTHER RESEARCH

The presented results are preliminary. Aiming to compute only a subset of ${}^k\mathbf{x}$ with the greatest possible efficiency the path tracing approach marked the starting point for our research. For this reason the variants of the classical Gauss algorithm, namely the branch-free variants of the LU and Gauss-Jordan algorithms, have not been ranked yet.

Analytical computations require a new way of thinking. There is no immediate connection between the performance of an analytical solution and the performance of the algorithm that was used to generate the solution. First results on algebraic differentiation show that the computation of derivatives is possible at little additional cost. This is due to the fact that derivatives can be build by reusing subexpressions of the original analytic solution.

Although the resulting machine code is rather small the preparation of the analytical solutions consumes much memory. First experiences with a garbage collection are promising and show that the compile-time memory requirements can be reduced drastically. Further research will focus on improving the quality of the generated code and other fields of application.

REFERENCES

Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Croz, J. D., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. 1999. *LAPACK Users' Guide*. Philadelphia: SIAM, third ed.

Bolobás, B. 2001. *Random Graphs*. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 2nd ed.

Bruno, J. L., and Lassagne, T. 1975. The generation of optimal code for stack machines. *J. ACM*, *22*(3), 382–396.

Conway, J. H. 1971. *Regular Algebra and Finite Machines*. Mathematics Series. Chapman and Hall.

Davis, T. A. 2006. *Direct Methods for Sparse Linear Systems*. Fundamentals of Algorithms. SIAM.

George, A., and Liu, J. W. H. 1981. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice Hall Series in Computational Mathematics.

Golub, G. H., and van Loan, C. F. 1996. *Matrix Computations (Johns Hopkins Studies in Mathematical Sciences)*. The Johns Hopkins University Press, third ed.

Isermann, N., Weitzel, M., and Wiechert, W. 2004. Kleene's theorem and the solution of metabolic carbon labeling systems. In *German Conference on Bioinformatics*, vol. 53 of *LNI*, (pp. 75–84). GI.

Lehmann, D. J. 1977. Algebraic structures for transitive closure. *Theoretical Computer Science*, *4*(1), 59–76.

Muchnick, S. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann.

Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. 2007. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, third ed.

Weitzel, M., Wiechert, W., and Nöh, K. 2007. The topology of metabolic isotope labeling networks. *BMC Bioinf.*, *8*(315).

Wiechert, W. 2001. ^{13}C metabolite flux analysis. *Metababolic Engineering*, *3*, 195–206.

Wiechert, W., and Wurzel, M. 2001. Metabolic isotopomer labeling systems. Part I: Global dynamic behaviour. *Mathematical Biosciences*, *169*, 173–205.

Zhang, F. (Ed.) 2005. *The Schur Complement and Its Applications (Numerical Methods and Algorithms)*. Numerical Algorithms. Springer Science+Business Media Inc.