LIBNMF – A LIBRARY FOR NONNEGATIVE MATRIX FACTORIZATION

Andreas JANECEK, Stefan SCHULZE GROTTHOFF
Wilfried N. GANSTERER

University of Vienna, Austria
Faculty of Computer Science
Research Lab Computational Technologies and Applications
Lenaugasse 2/8
1080-Vienna, Austria
e-mail: {andreas.janecek, wilfried.gansterer}@univie.ac.at

Abstract. We present *libNMF* – a computationally efficient high performance library for computing nonnegative matrix factorizations (NMF) written in *C*. Various algorithms and algorithmic variants for computing NMF are supported. *libNMF* is based on external routines from BLAS (Basic Linear Algebra Subprograms), LA-Pack (Linear Algebra package) and ARPack, which provide efficient building blocks for performing central vector and matrix operations. Since modern BLAS implementations support multi-threading, *libNMF* can exploit the potential of multi-core architectures. In this paper, the basic NMF algorithms contained in *libNMF* and existing implementations found in the literature are briefly reviewed. Then, *libNMF* is evaluated in terms of computational efficiency and numerical accuracy and compared with the best existing codes available. *libNMF* is publicly available at http://rlcta.univie.ac.at/software.

1 INTRODUCTION

Low-rank approximations of data (e.g., based on the singular value decomposition) have proven very useful in various data mining applications. Nonnegative matrix fac-

torization (NMF, cf. [22, 27]) leads to special low-rank approximations which satisfy non-negativity constraints. Non-negativity may improve interpretability and sparseness of the low-rank approximations. In general, the NMF approximates a given nonnegative matrix $A \in \mathbb{R}^{m \times n}$ by two nonnegative factor matrices $W \in \mathbb{R}^{m \times k}$ and $H \in \mathbb{R}^{k \times n}$, where $k \ll \min\{m, n\}$ is the rank of the approximation $WH \approx A$. Although the NMF is not unique and in general converges only to local minima, it has been shown that even a relatively low approximation quality can achieve acceptable classification accuracy in data mining applications [21].

The goal of this paper is to present and introduce a software library called *lib-NMF* that provides efficient implementations of several NMF routines. It contains various state-of-the-art NMF algorithms for computing NMF and methods for initializing the NMF factors W and H in order to speed up convergence. *libNMF* is purely written in C and allows for manually setting every parameter relevant for the calculation of NMF, but also offers default values for non-expert users. The library calls external routines from the software libraries LAPack [1] and ARPack [24], and is based on the BLAS (Basic Linear Algebra Subprograms). The routines use double-precision floating-point arithmetic. For some algorithms, single-precision versions are also provided. The documented source code of *libNMF*, some sets of test data, and a detailed documentation of the library are publicly available at `http://rlcta.univie.ac.at/software`.

Related work. In the last decade, many publications focused on improving, adapting, extending and re-designing algorithms for computing NMF. Various codes for computing NMF can be found in the literature. Table 1 provides a summary of important sources of publicly available code for computing NMF. The majority of available NMF code is written in Matlab. The function `nnmf.m` included in the Matlab Statistics Toolbox [33] since Matlab's R2008a release is probably one of the most widely used NMF codes. This function implements two of the original NMF algorithms – multiplicative update (MU) and alternating least squares (ALS) – introduced in [22] and [27], respectively. Cemgil [5] provides a Matlab implementation of variational Bayes for Kullback-Leibler divergence based NMF. Cichocki et al. [6] provide Matlab toolboxes for computing NMF for signal processing and image processing. Their algorithms comprise MU, exponentiated gradient, projected gradient (PG), conjugate gradient, and quasi-Newton. The same authors provide Matlab code in their book [7] about nonnegative matrix/tensor factorization. Another Matlab NMF toolbox has been written by Hansen et al. [14]. This toolbox contains a collection of existing NMF algorithms such as MU, ALS, and PG [25], as well as a self-developed algorithm called ALSOBS. Like with ALS the negative elements are set to zero but all other elements are adjusted using a method called *optimal brain surgeon* (OBS, [15]).

Hoyer [16] provides a widely used Matlab package for performing a projected gradient algorithm with sparseness constraints. Basic NMF is extended by including an option for controlling the sparseness of the factors W and H explicitly. Kim

et al. [19] provide Matlab implementations of fast Newton-type NMF methods in two versions: One based on an exact least squares solver for applications that require high accuracy, and an inexact implementation, which uses heuristics to solve the least squares problem in order to reduce computational effort at each iteration. The latter is better suited if computational efficiency is more important than accuracy.

Another often cited Matlab package was written by Lin [25]. Two projected gradient methods for NMF are proposed: the ALSPG method uses projected gradient methods for solving the update steps of the ALS algorithm, and the second method aims at directly applying projected gradients to NMF. Schmidt et al. [30] provide Matlab codes for sparse NMF using least squares with block principal pivoting, as well as Bayesian NMF.

Aside from Matlab, other NMF codes found in the literature are mainly written in R, Python or C/C++. Gaujoux [10] provides a framework for several NMF algorithms written in R, comprising several already published algorithms as well as an initialization method for W and H, called NNDSVD [4]. Liu [26] provides a similar framework in R, which is partly based on the codes available in [14]. Python codes for computing NMF are available in [25] and [31].Moreover, an interesting study investigating the performance of parallel NMF (written in Python) using OpenMP for shared-memory multi-core systems and CUDA for many-core graphics processors has been given in [2].

Dhillon et al. [8] provide a C++ library which contains several NMF algorithms and exploits the performance gains provided by optimized BLAS routines. The implemented algorithms comprise the basic MU algorithm (plus variants), variants of the ALS algorithm, a hybrid form of ALS and MU, as well as two NMF algorithms based on Bregman divergence as described in [8]. Green et al. [13] provide a C++ implementation of several NMF algorithms used for hierarchical clustering, and Pathak et al. [28] provide a generic NMF framework for the ITK toolkit (`http://www.itk.org`) – an open-source development framework for image segmentation and image registration programs. Wang et al. [34] provide C++ code for computing least squares nonnegative matrix factorization (LS-NMF).

Despite the fact that there are a large number of available NMF codes, so far there is no comprehensive, computationally efficient, well documented, and modularly structured library for computing NMF, with options to load/save all data involved in computing NMF, and with integrated initialization methods. The *libNMF* library presented in this paper is meant to be a first step in this direction. *libNMF* is freely available, computationally highly competitive with Matlab and other codes in high level languages, and considerably faster than Matlab clones, or codes written in R or Python. Compared to the C++ library [8], *libNMF* tends to be faster for comparable algorithms (similar performance for MU, significantly faster for ALS), and important additional algorithms (such as ALSPG and PG) are included. A detailed comparison of *libNMF* with the most competitive codes found in the literature is given in Section 4.2.

Authors	Ref.	Language	URL for software
The Mathworks	[3, 33]	Matlab	`http://www.mathworks.com/access/helpdesk/help/toolbox/stats/nnmf.html`
Cemgil	[5]	Matlab	`http://www.cmpe.boun.edu.tr/~cemgil/bnmf`
Cichocki et al.	[6]	Matlab	`http://www.bsp.brain.riken.jp/ICALAB/nmflab.html`
Cichocki et al.	[7]	Matlab	`http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0470746661.html`
Hansen et al.	[14]	Matlab	`http://isp.imm.dtu.dk/toolbox/nmf/index.html`
Hoyer	[16]	Matlab	`http://www.cs.helsinki.fi/u/phoyer/software.html`
Kim et al.	[19]	Matlab	`http://userweb.cs.utexas.edu/users/dmkim/Source/software/nnma/index.html`
Lin	[25]	Matlab/Python	`http://www.csie.ntu.edu.tw/~cjlin/nmf/index.html`
Schmidt et al.	[30]	Matlab	`http://mikkelschmidt.dk/index.php?id=2`
Gaujoux	[10]	R	`http://cran.r-project.org/web/packages/NMF/index.html`
Liu	[26]	R	`http://cran.r-project.org/web/packages/NMFN/index.html`
Battenberg et al.	[2]	Python	`http://www.eecs.berkeley.edu/~ericb`
Schmitt et al.	[31]	Python	`http://www.procoders.net/?p=409`
Dhillon et al.	[8]	C++	`http://www.kyb.mpg.de/bs/people/suvrit/work/progs/nnma.html`
Greene et al.	[13]	C++	`http://mlg.ucd.ie/nmf`
Pathak et al.	[28]	C++	`http://www.insight-journal.org/browse/publication/152`
Wang et al.	[34]	C++	`http://www.biomedcentral.com/1471-2105/7/175`

Table 1. Overview of publicly available NMF codes

Notation. In this article a matrix is represented by an uppercase italic letter (example: A, B, Σ, ...). A vector is represented by a lowercase bold letter (example: $\mathbf{u}$, $\mathbf{x}$, $\mathbf{q}_1$, ...). A scalar is represented by a lowercase Greek letter (example: λ, μ, ...). Matrix-matrix multiplications are denoted by "$*$" and element-wise multiplications by "$\cdot$".

Synopsis. In Section 2 we review some basics of NMF and discuss important NMF algorithms and variants. In Section 3 we introduce our *libNMF* library and discuss the implemented routines for algorithms mentioned in Section 2. Experimental evaluation of *libNMF* is summarized in Section 4, and in Section 5 we conclude our work and summarize ongoing and future research activities in this area.

2 REWIEV OF NMF

The nonnegative matrix factorization (NMF, cf. [27, 22]) consists of reduced rank *nonnegative* factors $W \in \mathbb{R}^{m \times k}$ and $H \in \mathbb{R}^{k \times n}$ with (problem dependent) $k \ll \min\{m, n\}$ that approximate a given nonnegative data matrix $A \in \mathbb{R}^{m \times n}$: $A \approx WH$. The nonnegativity constraints require that *all entries* in A, W and H are zero or positive. Although the product WH is only an *approximate* factorization of A of rank at most k, WH is called a nonnegative matrix factorization of A. The non-linear optimization problem underlying NMF can generally be stated as

$$\min_{W,H} f(W, H) = \frac{1}{2} ||A - WH||_F^2, \qquad (1)$$

where $||.||_F$ is the Frobenius norm ($||A||_F = (\sum |a_{ij}|^2)^{1/2}$). Although the Frobenius norm is commonly used to measure the error between the original data A and WH, other measures are also possible, for example, an extension of the Kullback-Leibler divergence to positive matrices [8], a convergence criterion based on the Karush-Kuhn-Tucker (KKT) conditions [20], or an angular measure based on the angle θ_i between successive basis vectors $W_i^{(t+1)}$ and $W_i^{(t)}$ [21]. A survey of distance measures for NMF can be found in [38]. Unlike the SVD, the NMF is not unique, and convergence is not guaranteed for all NMF algorithms. If they converge, then usually only to local minima (potentially different ones for different algorithms). Fortunately, the data compression achieved with only local minima has been shown to be of significant quality for many data mining applications [21, 17].

Due to its non-negativity constraints, NMF produces so-called "additive parts-based" (or "sum-of-parts") representations of the data (in contrast to many other representations such as SVD, PCA or ICA). This is an important benefit of NMF, since it makes the interpretation of the NMF factors much easier than for factors containing positive and negative entries, and enables NMF a non-subtractive combination of parts to form a whole [22]. For example, the features in W (called "basis vectors") may be topics of clusters in textual data, or parts of faces in image data.

Another favorable consequence of the nonnegativity constraints is that both factors W and H are often naturally sparse.

Initialization. Algorithms for computing NMF are generally iterative and require initialization of W and/or H. In the literature, random initialization of W and H is mostly used. In this case it may be necessary to run several instances of the algorithm. Moreover, algorithms based on random initialization are likely to suffer from slow convergence. It has been shown that better initialization strategies can lead to improvements in terms of faster convergence and faster error reduction. Beside classical initialization strategies based on k-means clustering techniques [36, 37], there are initialization techniques based on two successive SVD processes called NNDSVD (Nonnegative Double Singular Value Decomposition, [4]), and techniques based on efficient feature subset selection techniques [17, 18].

2.1 Algorithms for Computing NMF

Most existing NMF algorithms in the literature can be assigned to one of three general classes: multiplicative update (MU), alternating least squares (ALS) and projected gradient (PG) algorithms. A review of these three classes can be found, for example, in [3, 7, 25]. Pseudocode for the general structure of all NMF algorithms is given in Algorithm 1.

Algorithm 1 – General structure of NMF algorithms.

given matrix $A \in \mathbb{R}^{m \times n}$ and $k \ll \min\{m, n\}$:
for $rep = 1$ to *maxrepetition* **do**
  $W = \text{rand}(m, k);$
  $[H = \text{rand}(k, n);]$
  for $t = 1$ to *maxiter* **do**
    update W and H
    check termination criterion
  end for
end for

The variable *maxrepetition* specifies the number of repetitions of the complete algorithm for the case of randomly initialized W and H. Most algorithms need factors W and H both pre-initialized, but some algorithms (e.g., the ALS algorithm) only need one pre-initialized factor. In each repetition, NMF update steps are processed iteratively until a maximum number of iterations is reached (*maxiter*). The different update steps for the three basic NMF algorithms are briefly summarized in the following. If the approximation error of the algorithm drops below a predefined threshold, or if the change between two successive iterations is very small,

the algorithm may terminate before *maxiter* iterations are processed (for details, see Section 2.2).

2.1.1 Multiplicative Update (MU) Algorithm

The update steps for the original MU algorithm given in [23] are based on the mean squared error objective function. The update in each iteration consists of multiplying the current factors by a measure of the quality of the current approximation. The parameter ε in each iteration is often used to avoid division by zero. Following [29], a typical value used in practice is $\varepsilon = 10^{-9}$.

$$H^{(t+1)} = H^{(t)} \cdot \frac{\left(W^{(t)\top} * A\right)}{\left(W^{(t)\top} * W^{(t)}\right) * H^{(t)} + \varepsilon} \tag{2}$$

$$W^{(t+1)} = W^{(t)} \cdot \frac{\left(A * H^{(t+1)\top}\right)}{W^{(t)} * \left(H^{(t+1)} * H^{(t+1)\top}\right) + \varepsilon} \tag{3}$$

The divisions in Equations (2) and (3) are to be performed *element-wise*. Comments about the convergence of the MU algorithm can be found, for example, in [3, 11, 25].

2.1.2 Alternating Least Squares (ALS) Algorithms

Alternating least squares algorithms have been used and improved in several studies such as [20, 21, 27]. All ALS algorithms have in common that alternatively one factor (either W or H) is fixed, and the other one is minimized under corresponding constraints. In most algorithms, negative elements resulting in the process are set to 0 to ensure non-negativity.

Basic ALS algorithm. The basic ALS algorithm only needs to initialize one factor (W or H), the other factor is computed in the first iteration. In an alternating manner, a least squares step is followed by another least squares step. Typical implementations of ALS algorithms (see, for example, the implementation included in the Matlab Statistics Toolbox [33]) proceed as follows:
First, solve for $H^{(t+1)}$:

$$W^{(t)} * H^{(t+1)} = A \tag{4}$$

such that $f(W^{(t)}, H^{(t+1)}) \leq f(W^{(t)}, H^{(t)})$, and set all negative elements in $H^{(t+1)}$ to 0. Then solve for $W^{(t+1)}$:

$$H^{(t+1)} * W^{(t+1)} = A^{(t)} \tag{5}$$

such that $f(W^{(t+1)}, H^{(t+1)}) \leq f(W^{(t)}, H^{(t+1)})$. Some studies, for example [11, 25], have analyzed the convergence properties of ALS algorithms. It has been proven that ALS will converge to a fixed point which may be a local extremum or a saddle point (cf. [21]). The solution of Equations (4) and (5) can, for example, be computed using a QR-factorization or an LU-factorization, or based on computing the pseudo inverse of H and W.

Normal equations ALS algorithm. This variant of the ALS algorithm has computational advantages over the implementation included in the Matlab Statistics Toolbox [33] in some cases – especially if $k \ll \min\{m, n\}$. This variant which we call NEALS (normal equations alternating least squares) changes Equations (4) and (5) to:

$$\left(W^{(t)\top} * W^{(t)}\right) * H^{(t+1)} = W^{(t)\top} * A, \tag{6}$$

$$\left(H^{(t+1)\top} * H^{(t+1)}\right) * W^{(t+1)} = H^{(t+1)\top} * A. \tag{7}$$

In exact arithmetic, there is no difference between ALS and NEALS. NEALS has some numerical disadvantages compared to ALS because of the squaring of the condition numbers. However, in the given context NEALS can be very attractive for various reasons:

1. Since in many cases we need to compute a *low rank* NMF with relatively small k, the resulting normal equations are much smaller than the least squares formulations of basic ALS.

2. The additional expense is the matrix multiplication required for forming the normal equations. However, it is well known that this operation has a favorable computation per communication ratio and thus can be well mapped onto modern multi-core architectures.

3. The potential loss in numerical accuracy is usually not too severe, because the computation of an NMF is only an approximation anyway.

2.1.3 Projected Gradient Algorithms

This third group of algorithms is based on the idea to take a step in the direction of the negative gradient, the direction of the steepest descent (which can be computed using the partial derivatives for H and W). An interesting study investigating gradient descent algorithms was published by Lin [25]. In this paper, the author proposed the use of a projected gradient bound-constrained optimization method for computing the NMF in two situations: by solving the alternating nonnegative least squares problems with projected gradient methods, and by directly minimizing the objective function in Equation (1) using projected gradients.

ALS using Projected Gradient (ALSPG) Algorithm. Here, the projected gradient is used to solve the nonnegative least squares problem discussed in Section 2.1.2. Analogously to ALS, one factor (W or H) is updated while A and the other factor are kept constant. The general update steps look as follows:

$$H^{(t+1)} = H^{(t)} - \alpha_H \nabla_H f\left(W^{(t)}, H^{(t)}\right) \tag{8}$$

$$W^{(t+1)} = W^{(t)} - \alpha_W \nabla_W f\left(W^{(t)}, H^{(t+1)}\right). \tag{9}$$

α_H and α_W are step-size parameters which have to be chosen carefully in order to get a good approximation (cf. the discussion in [25]). The partial derivatives in Equations (8) and (9) are $\nabla_H f(W^{(t)}, H^{(t)}) = W^{(t)\top}(W^{(t)}H^{(t)} - A)$ and

$\nabla_W f(W^{(t)}, H^{(t+1)}) = (W^{(t)}H^{(t+1)} - A)H^{(t+1)\top}$, respectively. Experiments in [25] show that this method is computationally very competitive and in many cases has better convergence properties than the standard MU approach.

Direct Projected Gradient (PG) Algorithm. In this algorithm projected gradient methods are used to directly minimize the objective function in Equation (1). From the current solution $(W^{(t)}, H^{(t)})$, both matrices are simultaneously updated to $(W^{(t+1)}, H^{(t+1)})$ in the general form:

$$(W^{(t+1)}, H^{(t+1)}) = (W^{(t)}, H^{(t)}) - \alpha(\nabla_W f(W^{(t)}, H^{(t)}), \nabla_H f(W^{(t)}, H^{(t)})) \qquad (10)$$

2.2 Termination Criteria

Generally speaking, three termination criteria can be applied. The simplest convergence criterion which is used in almost all NMF algorithms is to run for a fixed number of iterations (cf. the parameter *maxiter* in Algorithm 1). Since the most appropriate value for *maxiter* is problem-dependent, this is not a mathematically appealing way to control the number of iterations, but applies when the required approximation accuracy does not drop below a pre-defined threshold. Another problem-dependent convergence criterion is the approximation accuracy of the NMF objective function, which obviously depends on the size and structure of the data but may be useful to compare the approximation accuracy of different algorithms. As already mentioned, different convergence measures can be applied, such as the Frobenius norm (see Equation (1)), Kullback-Leibler divergence, KKT, or angular measures. The relative change of factors W and H from one iteration to the next iteration is the basis for another convergence criterion. If this change is below a pre-defined threshold δ, the algorithm is terminated. Depending on the NMF algorithm used, additional termination criteria may apply (e.g., time limit, change of the projection norm for projected gradient methods, etc.).

3 LIBNMF

In this section we present the first public version (version 1.02) of our *libNMF* library, summarize general characteristics, and discuss its usage. Then we focus on the computational routines implemented in *libNMF*.

3.1 General Notes

We assume that matrices are stored in two-dimensional arrays. To simplify the usage of Fortran high performance routines (e.g., from LAPack), arrays are logically accessed in column-major order, which is how Fortran accesses two-dimensional arrays. Unless stated otherwise, all routines use IEEE double precision floating-point arithmetic.

3.1.1 External Libraries

libNMF utilizes routines from BLAS and LAPack. The NNDSVD initialization
(`LIBNMF/nndsvd`, cf. Section 3.2.6) requires computation of SVDs, which is done
using ARPack routines [24].

3.1.2 Subroutines

The BLAS, LAPack, and ARPack routines utilized by *libNMF* are listed together
with their functionality in Table 2.

Double precision BLAS routines	
`BLAS/daxpy`	calculating $y = a * x + y$
`BLAS/dcopy`	copying a vector to another vector
`BLAS/dgemm`	matrix-matrix multiplication
`BLAS/dgemv`	matrix-vector multiplication
`BLAS/dlamch`	determining machine precision epsilon
`BLAS/dscal`	scaling a vector by a constant
Double precision LAPack routines	
`LAPack/dgeqp3`	QR-factorization with column pivoting
`LAPack/dgesv`	solving system of linear equations (LU)
`LAPack/dlacpy`	copying a matrix to another matrix
`LAPack/dlange`	calculating the Frobenius norm
`LAPack/dorgqr`	generating economy sized explicit Q in QR factorization
`LAPack/dtrtrs`	solving a triangular system
ARPack routines for `LIBNMF/nndsvd`	
`ARPack/dsaupd`	implicitly restarted Arnoldi iteration
`ARPack/dseupd`	post-processing routine for large-scale symmetric eigenvalue calculation

Table 2. BLAS, LAPack and ARPack routines used in *libNMF*

3.2 Computational Routines

The main routines included in version 1.02 of *libNMF* are discussed briefly in the
following. In the next versions additional routines will be added to *libNMF* in order
to cover a wider spectrum of different NMF algorithms.

3.2.1 `LIBNMF/nmf_mu`

This routine implements the multiplicative update algorithm as described in Sec-
tion 2.1.1. Each matrix-matrix multiplication is calculated by calling `BLAS/dgemm`,
and all element-wise operations ($\cdot$, $+$, and the division) are calculated in a for-loop.

　　With increasing number of iterations the number of very small positive and
zero entries increases in both factor matrices W and H. Performance tests showed

that this leads to an increase of runtime per iteration. Therefore small positive entries (in the order of machine precision) are set to zero explicitly in every iteration. Moreover, experiments showed that also an increasing number of zero entries slowed down `LIBNMF/nmf_mu`. This effect could be almost completely compensated by checking whether the result will be zero and in that case directly setting it instead of computing it.

The routine `LIBNMF/nmf_mu_singleprec` implements a single precision version of `LIBNMF/nmf_mu`, using the single precision versions of BLAS and LAPack routines (i.e., `BLAS/sgemm` instead of `BLAS/dgemm`.)

3.2.2 `LIBNMF/nmf_als`

This routine implements an ALS algorithm as described in Section 2.1.2. For calculating $H^{(t+1)}$, first a QR-factorization (`LAPack/dgeqp3`) with column pivoting of $W^{(t)}$ is computed, resulting in an explicit representation of R and an implicit representation of Q. Then, the upper triangular sub-block of $R \in \mathbb{R}^{k \times k}$ is copied (`LAPack/dlacpy`) and an economy sized explicit $Q \in \mathbb{R}^{m \times k}$ is computed (`LAPack/dorgqr`). After calculating $U = Q^\top * A \in \mathbb{R}^{k \times n}$ (`BLAS/dgemm`), the equation $R * H^{(t+1)} = U$ is solved (`LAPack/dtrtrs`). Finally, the rows of $H^{(t+1)}$ are permuted according to the pivoting of the factorization (`BLAS/dcopy`), and negative entries are set to zero. Based on $H^{(t+1)}$, $W^{(t+1)}$ is calculated in the next step using a QR-factorization with column pivoting of $H^{(t+1)\top}$.

3.2.3 `LIBNMF/nmf_neals`

This routine implements the NEALS algorithm from Section 2.1.2. For calculating $H^{(t+1)}$, first two auxiliary matrices are calculated (`BLAS/dgemm`): $T_1 = W^{(t)\top} * W^{(t)} \in \mathbb{R}^{k \times k}$ and $T_2 = W^{(t)\top} * A \in \mathbb{R}^{k \times n}$. Then, an LU-factorization (`LAPack/dgesv`) is used to solve the equation $T_1 * H^{(t+1)} = T_2$ for $H^{(t+1)}$, and negative elements are set to zero.

For calculating $W^{(t+1)}$, $T_3 = H^{(t+1)} * H^{(t+1)\top} \in \mathbb{R}^{k \times k}$ and $T_4 = H^{(t+1)} * A^\top \in \mathbb{R}^{k \times m}$ are calculated. Then the equation $T_3 * W^{(t+1)\top} = T_4$ is solved for $W^{(t+1)}$ using an LU-factorization, and negative elements are set to zero.

3.2.4 `LIBNMF/nmf_alspg`

This routine implements the ALSPG algorithm as proposed in [25]. Prior to the iterative update steps initial gradients are calculated (using three `BLAS/dgemm` calls):

$$
\begin{aligned}
\nabla_H &= W^{(0)} * (H^0 * H^{(0)\top}) - A * H^{(0)\top} & (11)\\
\nabla_W &= (W^{(0)\top} * W^{(0)}) * H^{(0)} - W^{(0)\top} * A. & (12)
\end{aligned}
$$

Moreover, the norm of the initial gradients is calculated (`LAPack/dlange`), which is used as an additional stopping condition for projected gradient algorithms (cf. Section 2.2).

Update steps. In every iteration, first the new projection norm is calculated, then $W^{(t+1)}$ and $H^{(t+1)}$ are updated alternately. Update steps are computed in a separate routine called `LIBNMF/pg_subprob` which is also used by the PG algorithm (cf. Section 3.2.5), and briefly discussed in the following.

`LIBNMF/pg_subprob`.   First, two auxiliary matrices $T_5 = W^\top * A$ and $T_6 = W^\top * W$ are calculated (`BLAS/dgemm`).Then, two nested loops are run. In the outer loop the new gradient $\nabla = T_5 * H - T_6$ is calculated (`LAPack/dlacpy`, `BLAS/dgemm`). In the inner loop the step-size parameters α and β are determined based on the change from the current solution to the newly computed solution (`LAPack/ dlacpy`, `BLAS/daxpy`, `BLAS/dgemm`).

3.2.5 `LIBNMF/nmf_pg`

This routine implements the projected gradient algorithm as proposed in [25]. In every iteration, gradients ∇_H and ∇_W are calculated similar to Equations (11) and (12). In the first iteration the initial gradient (using `LAPack/dlange`), and H (using `LIBNMF/pg_subprob`) are calculated. In all iterations (including the first iteration) the new projection norm is calculated and $W^{(t+1)}$ and $H^{(t+1)}$ are updated, again using a step-size parameter which is calculated in an inner loop (`LAPack/dlacpy`, `BLAS/daxpy`).

3.2.6 `LIBNMF/nndsvd`

As a first initialization strategy (cf. Section 2) for W and H we implemented the Nonnegative Double Singular Value Decomposition (NNDSVD) technique as described in [4]. It is based on two SVD processes – one approximating the data matrix A (rank-k approximation), the other approximating positive sections of the resulting partial SVD factors (`BLAS/dgemm`, `ARPack/dsaupd`, `ARPack/dseupd`).

4 EXPERIMENTAL EVALUATION

We performed detailed experiments to evaluate the performance of the routines in *libNMF*. First, we briefly discuss the setup (data sets and hardware) used to measure the runtimes. Then, we discuss some performance issues of existing NMF codes found in the literature. Finally, we provide a runtime comparison of *libNMF* with Matlab implementations of identical algorithms as well as with representative NMF codes found in the literature written in different programming languages.

4.1 Experimental Setup

We used the *p53 Mutants* dataset from the UCI machine learning repository (publicly available at `http://archive.ics.uci.edu/ml`). It consists of 16 772 instances described by 5 409 attributes (including a binary class attribute which separates the

instances into two groups "actives/inactives"). 180 instances with missing values were deleted and the remaining instances were separated into a training set consisting of the first 75 % active instances and the first 75 % inactive instances, and a test set consisting of the remaining 25 % of each group.

Hardware. All runtimes were measured on a SUN FIRE X4600 M2 with 8 AMD Opteron 8 356 quad-core processors with 3.2 GHz, 2 MB L3 cache, and 32 GB of main memory (DDR-II 666). CPUs are connected to each other by a HyperTransport link running at 8 GB/second.

Software. As already mentioned in Section 1, most of the available NMF codes are written in Matlab.In the last years, the support of multithreaded computations for several linear algebra operations included in Matlab has been improved, and Matlab version 2010a shows noteworthy speedup in several cases (e.g., matrix multiply, linear equations, etc. [32]) compared to older Matlab versions. Matlab efficiently utilizes BLAS and thus also achieves a good performance for most NMF algorithms.

However, in many circumstances it may not be efficient or not feasible to use a commercial software product like Matlab. Matlab clones, such as Octave, O-Matrix or Scilab, are significantly cheaper or even free, but usually cannot compete with Matlab in term of computational efficiency, number of available routines, support, usability, etc. NMF implementations in O-Matrix (`http://www.omatrix.com`) showed competitive runtimes compared to current Matlab routines in our experiments (e.g., the matrix multiply routine which is essential for NMF is even slightly faster than with Matlab 2010a), but O-Matrix is only available for Windows. Scilab (`http://www.scilab.org`) and Octave (`http://www.gnu.org/software/octave`) are also available for Unix-like systems and can also be built with optimized BLAS routines. However, experiments with Scilab and Octave on several machines with the outcome that overall both programs could not compete with Matlab in terms of runtime performance. Scilab also showed severe memory allocation problems when large matrices were used. Moreover, for Scilab the Matlab files need to be converted to Scilab files, which works smoothly for simple code-fragments but is often more difficult for complex code.

Other available NMF codes are written in R [10, 26] or Python [31]. Several benchmarks (e.g., `http://mlg.eng.cam.ac.uk/dave/rmbenchmark.php`) as well as our own evaluation showed that R is generally slower than Matlab for matrix operations, which are an essential part of all NMF algorithms. Comparisons of runtimes with Python codes from [31] are given in Section 4.2. The Python modules announced in [2] were not available at the time when this paper was written.

NMF codes written in C/C++ are among the fastest if they are based on BLAS and LAPack routines. However, not all available C/C++ codes are directly comparable to *libNMF*. For example, [13] use NMF to compute an ensemble clustering algorithm based on the symmetric NMF algorithm as proposed in [9]. Since this al-

gorithm works only for a symmetric matrix A and creates ensembles of NMF instead
of single factorizations the achieved runtimes are not directly comparable to *libNMF*.
The code from [28] was written to be used within an image processing toolkit, which
makes it difficult to compare it to *libNMF*. Moreover, it is not possible to compile
the code separately without linking the complete toolkit. The parallel code from [34]
is based on LAM/MPI and was intended for use on Beowulf clusters. We compiled
the desktop version of the code (which is also available), however, this version does
not support the linking of BLAS routines.

We report detailed runtime comparisons of *libNMF* routines with

- Matlab implementations, in particular of the algorithms implemented in [33]
  (MU, ALS), in [25] (ALSPG, PG), of the NEALS algorithm as discussed in
  Section 2.1.2, and of the NNDSVD initialization from Section 3.2.6. Moreover,
  we measured the runtime needed to achieve a given accuracy for algorithms
  from [14, 16, 20, 30], which are not yet implemented in *libNMF*.

- C++ implementations, in particular of the MU and the ALS algorithm im-
  plemented in [8] compiled with Goto BLAS and the GNU Scientific Library
  (`http://www.gnu.org/software/gsl`).

- Python implementations, in particular with the Python modules from [31] com-
  piled with Atlas BLAS version 3.9.23.

4.2 Runtime Comparison

The runtime comparisons are split up into two parts. In the first part, we com-
pare the runtime of *libNMF* routines to implementations of *identical* algorithms
in Matlab (v2010a). In the second part, we compare *libNMF* routines to the *best*
algorithms/implementations found in the literature from a user's point-of-view. In
this setting, the "best" routines are those which are able to achieve a given accuracy
in the shortest amount of time. In our experiments we experimented with Atlas
BLAS version 3.8.3 and development version 3.9.11 [35], and with Goto BLAS ver-
sion 1.13 [12]. Overall, Goto BLAS seemed to be faster, setting an emphasis on
parallel performance. Atlas BLAS 3.8.3 seemed to utilize multiple CPU cores con-
siderably less than Goto BLAS, which improved in the newer version (which features
a new multithreading implementation). However, all *libNMF* runtimes in this paper
are based on Goto BLAS v1.13. Goto BLAS utilizes all 32 cores available on our
system.

Speedup over Matlab 2010a. Figures 1 and 2 show the speedup of *libNMF*
routines over Matlab routines implementing identical NMF algorithms (i.e., the
results after each iteration are numerically identical, there is only a difference in
runtime) for varying rank k, using randomly initialized factors W and H. In order
to investigate the runtimes for varying shapes of rectangular data sets, we truncated
the larger dimension of our dataset in steps of 2 000. As Figure 1 shows, *libNMF*
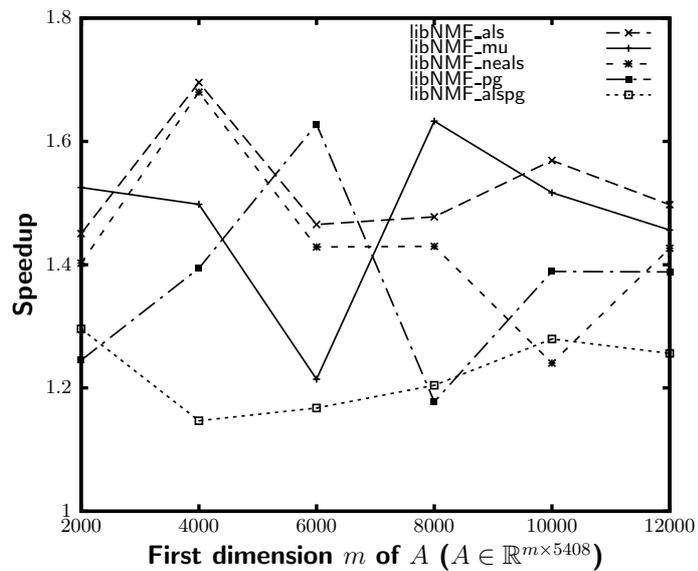
Fig. 1. Speedup over Matlab 2010a – $k = 50$
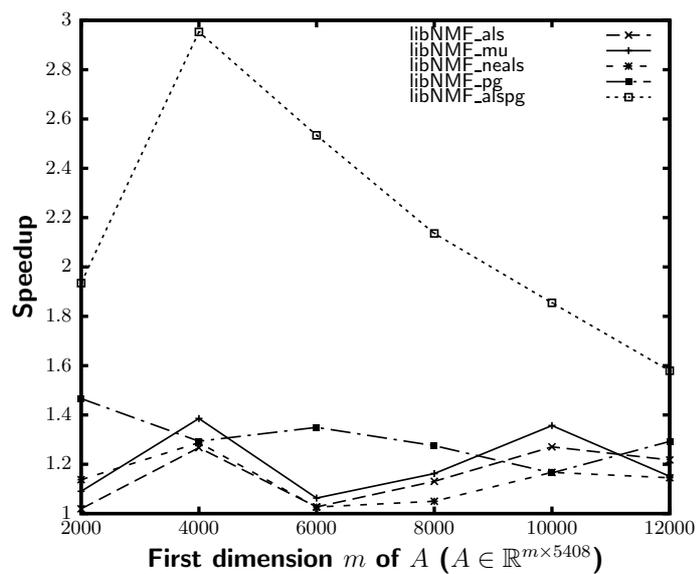
Fig. 2. Speedup over Matlab 2010a – $k = 10$

routines are always faster than corresponding Matlab routines – overall, a speedup of about 1.4 over Matlab 2010 could be achieved. It is interesting to note the higher speedup of `LIBNMF/nmf_alspg` with lower rank k in Figure 2, which differs from the behavior of the other algorithms. For rank $k = 10$, `LIBNMF/nmf_alspg` is on average more than twice as fast as the Matlab implementation.

For computing the NNDSVD initialization (cf. Section 3.2.6, not shown in Figures 1 and 2) *libNMF* achieved an impressive speedup of 24 over Matlab 2010a. However, this speedup needs to be considered carefully since Matlab's `svds` routine only utilized one core in our experiments.
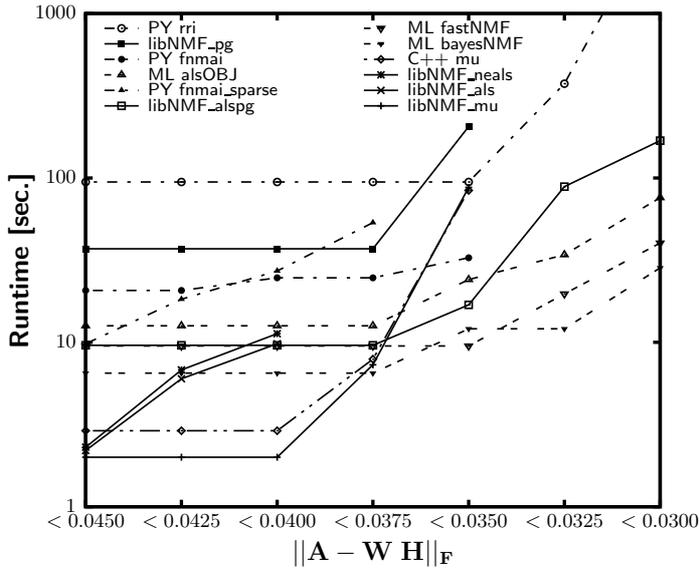

Fig. 3. Runtime for fixed accuracy (NNDSVD) – $k = 25$

Runtime for fixed accuracy. Figure 3 shows a runtime comparison of several implementations of NMF algorithms from a user's point-of-view for a $12\,000 \times 5\,408$ subset of the training set mentioned in Section 4.1. The runtime needed to achieve a given approximation accuracy (measured in the Frobenius norm) is plotted along the y-axis ($\log_{10}$ scale). Curves which do not continue indicate that the corresponding algorithm is not able to achieve a specific accuracy. The runtimes of the five *libNMF* algorithms are plotted together with the seven "best" NMF implementations found in the literature. All algorithms from [8, 14, 16, 20, 30, 31] that are not present in Figure 3 needed considerably longer time to reach a specific approximation accuracy. The runtimes are shown for pre-initialized factors W and H (using NNDSVD) and $k = 25$.

Figure 3 can be interpreted as follows: a fast and rough approximation (approximation error < 0.0450) can be achieved in about two seconds with the *libNMF* algorithms `LIBNMF/nmf_mu`, `LIBNMF/nmf_als`, or `LIBNMF/nmf_neals`, but only `LIBNMF/nmf_mu` is able to achieve even an approximation error < 0.0400 in the same time. If a better approximation is desired, other algorithms are faster than `LIBNMF/nmf_mu`. It turns out that the *fastNMF* and *bayesNMF* algorithms from [30], and the *alsOBJ* algorithm from [14] are the fastest ones for computing a close approximation of A – even faster than the best *libNMF* algorithm `LIBNMF/nmf_alspg`. Since algorithms implemented in *libNMF* are almost always faster than identical algorithms implemented in Matlab, we are currently working on integrating the algorithms from [14, 30] into *libNMF*.

5 CONCLUSION AND FUTURE WORK

We introduced a new library for computing nonnegative matrix factorization (NMF) called *libNMF*, which implements several computationally efficient NMF routines. *libNMF* is a modularly structured, open source library written in C which calls computationally efficient external libraries, such as BLAS, LAPack and ARPack. Runtime comparisons with Matlab version 2010a and other NMF codes found in the literature showed that *libNMF* achieves significant speedups over other implementations of identical algorithms.

Our experiments also revealed that some algorithms which are not yet integrated into *libNMF* achieve high approximation accuracy in shorter time. We are currently working on improvements and extensions of *libNMF*. Beside additional algorithms, such as *fastNMF* and *bayesNMF* from [30] or quasi-Newton algorithms, we plan to implement sparseness constraints [16], different error measures (such as the ones mentioned in Section 2), and support for other initialization strategies (e.g., [18]). Moreover, we plan to extend our library with NMF variants optimized for graphic processing units (GPUs). *libNMF* is available at `http://rlcta.univie.ac.at/software`.

Acknowledgments

REFERENCES

[1] ANDERSON, E.—BAI, Z.—BISCHOF, C. et al.: LAPACK Users' Guide. Society for Industrial and Applied Mathematics. Philadelphia, PA, third edition, 1999.

[2] Battenberg, E.—Wessel, D.: Accelerating Non-Negative Matrix Factorization for Audio Source Separation on Multi-Core and Many-Core Architectures. In Proc. of 10th Int. Society for Music Information Retrieval Conf., 2009, pp. 501–506.

[3] Berry, M. W.—Browne, M.—Langville, A. N.—Pauca, P. V.—Plemmons, R. J.: Algorithms and Applications for Approximate Nonnegative Matrix Factorization. Computational Statistics & Data Analysis, Vol. 52, 2007, No. 1, pp. 155–173.

[4] Boutsidis, C.—Gallopoulos, E.: SVD Based Initialization: A Head Start for Nonnegative Matrix Factorization. Pattern Recogn., Vol. 41, 2008, No. 4, pp. 1350–1362.

[5] Cemgil, A. T.: Bayesian Inference for Nonnegative Matrix Factorisation Models. Intell. Neuroscience, 2009, pp. 1–17.

[6] Cichocki, A.—Zdunek, R.—Amari, S.: Csiszr's Divergences for Non-Negative Matrix Factorization: Family of New Algorithms. LNCS, Vol. 3889, 2006, No. 1, pp. 32–39.

[7] Cichocki, A.—Zdunek, R.—Phan, A. H.—Amari, S.: Nonnegative Matrix and Tensor Factorizations: Applications to Exploratory Multi-Way Data Analysis and Blind Source Separation. Wiley 2009.

[8] Dhillon, I. S.—Sra, S.: Generalized Nonnegative Matrix Approximations with Bregman Divergences. Advances in Neural Information Processing Systems, Vol. 18, 2005, pp. 283–290.

[9] Ding, C.—He, X.—Simon, H. D.: On the Equivalence of Nonnegative Matrix Factorization and Spectral Clustering. In Proc. SIAM Data Mining Conf. 2005, pp. 606–610.

[10] Gaujoux, R.: Package NMF. 2010, `http://cran.r-project.org/web/packages/NMF`.

[11] Gonzales, E.—Zhang, Y.: Accelerating the Lee-Seung Algorithm for Nonnegative Matrix Factorization. Technical report, Department of Computational and Applied Mathematics, Rice University, 2005.

[12] Goto, K.—van de Geijn, R. A.: High-Performance Implementation of the Level-3 BLAS. ACM Trans. Math. Softw., Vol. 35, 2008, No. 1, pp. 1–18.

[13] Greene, D.—Cagney, D.—Krogan, N.—Cunningham, P.: Ensemble Non-Negative Matrix Factorization Methods for Clustering Protein-Protein Interactions. Bioinformatics, Vol. 24, 2008, No. 15, pp. 1722–1728.

[14] Hansen, L. K.: NMF: DTU Toolbox. 2006, `http://isp.imm.dtu.dk/toolbox/nmf/index.html`.

[15] Hassibi, B.—Stork, D.—Wolff, G.: Optimal Brain Surgeon and General Network Pruning. In IEEE International Conference on Neural Networks, pp. 293–299, 1993.

[16] Hoyer, P. O.: Non-Negative Matrix Factorization With Sparseness Constraints. Journal of Machine Learning Research, Vol. 5, pp. 1457–1469, 2004.

[17] Janecek, A. G. K.—Gansterer, W. N.: E-Mail Classification Based on NMF. In 9th SIAM International Conference on Data Mining 2009, Proceedings in Applied Mathematics, 3, SIAM, pp. 1345–1354, 2009.

[18] JANECEK, A. G. K.—GANSTERER, W. N.: Utilizing Nonnegative Matrix Factorization for E-Mail Classification Problems. In M. W. Berry and J. Kogan (Eds.): Survey of Text Mining III: Application and Theory, Wiley, 2010, pp. 57–80.

[19] KIM, D.—SRA, S.—DHILLON, I. S.: Fast Newton-Type Methods for the Least Squares Nonnegative Matrix Approximation Problem. In Proc. SIAM Data Mining Conf., pp. 343–354, 2007.

[20] KIM, H.—PARK, H.: Nonnegative Matrix Factorization Based on Alternating Nonnegativity Constrained Least Squares and Active Set Method. SIAM J. Matrix Anal. Appl., Vol. 30, 2008, No. 2, pp. 713–730.

[21] LANGVILLE, A. N.—MEYER, C. D.—ALBRIGHT, R.: Initializations for the Nonnegative Matrix Factorization. In SIGKDD '06: Proceedings of the 12th ACM International Conference on Knowledge Discovery and Data Mining, 2006.

[22] LEE, D. D.—SEUNG, H. S.: Learning Parts of Objects by Nonnegative Matrix Factorization. Nature, Vol. 401, 1999, No. 6755, pp. 788–791.

[23] LEE, D. D.—SEUNG, S. H.: Algorithms for Non-Negative Matrix Factorization. Advances in Neural Information Processing Systems, Vol. 13, 2001, pp. 556–562.

[24] LEHOUCQ, R. B.—SORENSEN, D. C.—YANG, C.: ARPACK Users Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods. SIAM, 1998.

[25] LIN, C. J.: Projected Gradient Methods for Nonnegative Matrix Factorization. Neural Comput., Vol. 19, 2007, No. 10, pp. 2756–2779.

[26] LIU, S.: Package NMFN. Available on-line: `http://cran.r-project.org/web/packages/NMFN/NMFN.pdf`, 2009.

[27] PAATERO, P.—TAPPER, U.: Positive Matrix Factorization: A Nonnegative Factor Model With Optimal Utilization of Error Estimates of Data Values. Environmetrics, Vol. 5, 1994, No. 2, pp. 111–126.

[28] PATHAK, S.—HAYNOR, D.—LAU, C.—HAWRYLYCZ, M.: Non-Negative Matrix Factorization Framework for Dimensionality Reduction and Unsupervised Clustering. The Insight Journal (open-source), `http://hdl.handle.net/1926/502`, 2007.

[29] PIPER, J.—PAUCA, V. P.—PLEMMONS, R. J.—GIFFIN, M.: Object Characterization from Spectral Data Using Nonnegative Factorization and Information Theory. In Proc. of Amos Technical Conf., pp. 591–600, 2004.

[30] SCHMIDT, M. N.—LAURBERG, H.: Non-Negative Matrix Factorization With Gaussian Process Priors. Comp. Intelligence and Neuroscience, 2008, No. 1, pp. 1–10.

[31] SCHMITT, U.: NNMA Toolbox. `http://www.procoders.net/?p=409`, 2008.

[32] The Mathworks. Matlab release notes. Available on-line, 2010.

[33] The Mathworks. Matlab statistics toolbox. Available on-line, 2010. `http://www.mathworks.com/products/statistics`.

[34] WANG, G.—KOSSENKOV, A. V.—OCHS, M. F.: LS-NMF: A Modified Non-Negative Matrix Factorization Algorithm Utilizing Uncertainty Estimates. BMC Bioinformatics, Vol. 7, 2006, No. 175, pp. 1–10.

[35] WHALEY, R. C.—PETITET, A.: Minimizing Development and Maintenance Costs in Supporting Persistently Optimized BLAS. Software: Practice and Experience, Vol. 35, 2005, No. 2, pp. 101–121.

[36] Wild, S. M.: Seeding Non-Negative Matrix Factorization With the Spherical *k*-Means Clustering. Master's thesis, University of Colorado, 2002.

[37] Wild, S. M.—Curry, J. H.—Dougherty, A.: Improving Non-Negative Matrix Factorizations Through Structured Initialization. Pattern Recog., Vol. 37, 2004, No. 11, pp. 2217–2232.

[38] Xue, Y.—Tong, C. S.—Zhang, W.: Survey of Distance Measures for nmf-Based Face Recognition. In International Conference on Computational Intelligence and Security, pp. 1039–1049, 2007.

Andreas Janecek received his Ph. D. degree in Computer Science in 2010, and his M. Sc. degree in Business Informatics in 2005, both from the University of Vienna, Austria. His research activities include data mining and machine learning algorithms, with a focus on high performance and distributed computing aspects of these techniques, and computational intelligence such as population based algorithms and evolutionary computing. He is currently a post-doctoral researcher at the School of Electronic Engineering and Computer Science, Beijing University, China.

Stefan Schulze Grotthoff is currently studying Computer Science in the Bachelor program at the University of Vienna, Austria. His research activities revolve around nonnegative matrix factorizations and their application for data classification.

Wilfried N. Gansterer is an Assistant Professor of Computer Science at the University of Vienna, Austria. He received a Ph. D. degree in Scientific Computing from Vienna University of Technology, Austria, in 2000, a Master's degree in Scientific Computing/Computational Mathematics from Stanford University, USA, in 1996, and a Master's degree in Mathematics from Vienna University of Technology in1994. His research interests are in scientific and high performance computing, parallel and distributed computing, data mining and machine learning algorithms, and in related application problems in computational life sciences and internet security.