Real-Time Principal Component Pursuit

Graeme Pope*, Manuel Baumann*, Christoph Studer†, and Giuseppe Durisi‡

*Dept. of Information Technology and Electrical Engineering, ETH Zurich, Zurich, Switzerland
e-mails: gpope@nari.ee.ethz.ch, manubaum@ee.ethz.ch

†Dept. of Electrical and Computer Engineering, Rice University, Houston, TX, USA
e-mail: studer@rice.edu

‡Dept. of Signals and Systems, Chalmers University of Technology, Gothenburg, Sweden
e-mail: durisi@chalmers.se

***Abstract*—Robust principal component analysis (RPCA) deals with the decomposition of a matrix into a low-rank matrix and a sparse matrix. Such a decomposition finds, for example, applications in video surveillance or face recognition. One effective way to solve RPCA problems is to use a convex optimization method known as principal component pursuit (PCP). The corresponding algorithms have, however, prohibitive computational complexity for certain applications that require real-time processing. In this paper we propose a variety of methods that significantly reduce the computational complexity. Furthermore, we perform a systematic analysis of the performance/complexity tradeoffs underlying PCP. For synthetic data, we show that our methods result in a speedup of more than 365 times compared to a reference C implementation at only a small loss in terms of recovery error. To demonstrate the effectiveness of our approach, we consider foreground/background separation for video surveillance, where our methods enable real-time processing of a 640×480 color video stream at 12 frames per second (fps) using a quad-core CPU.**

I. Introduction

Robust principal component analysis (RPCA) deals with the problem of making PCA (a widespread tool for the analysis of high-dimensional data [1]) robust to outliers and grossly corrupted observations [2], [3]. The RPCA problem can be formulated mathematically as the problem of decomposing a matrix consisting of the sum of a low-rank matrix and a sparse matrix into these two components, without prior knowledge of the low-rank part nor of the sparsity pattern of the sparse part [3]. One effective way to solve this decomposition problem, which is in general intractable [4], [5], is to perform a convex relaxation [3], [4]. The resulting computationally-tractable optimization problem is known as *principal component pursuit* (PCP) [3]. Under certain conditions on the rank and the sparsity level of the two components, PCP is able to *exactly* recover both components with high probability [3], [4]. Furthermore, PCP can be reformulated as a semidefinite program [4], for which several standard solvers based on interior-point methods are available (see, e.g., [6]). Unfortunately, these solvers fail to efficiently handle matrices of large size, which are typically encountered in the applications where PCA is used. To overcome this issue, several algorithms based on first-order methods [7], have been recently proposed in the literature, e.g., iterative-thresholding, augmented Lagrangian multipliers (ALM), and accelerated proximal gradient (see [2] and references therein). However, for certain applications, the resulting algorithms still have prohibitive computational complexity. For example, the ALM algorithm investigated in [3] in the context of foreground/background separation for video surveillance, requires roughly 43 minutes to converge on a desktop PC, when applied to the (25 344×200)-dimensional matrix obtained through the concatenation of 200 video frames, each consisting of 176×144 pixels. Such a slow convergence rate on state-of-the-art CPUs clearly prevents such algorithms from being used in real-time applications.

Contributions: In this paper, we perform a simulative performance and complexity analysis for the ALM algorithm. To achieve real-time processing on state-of-the-art CPUs, we propose a variety of techniques, which result in significant complexity savings compared to a reference implementation in C using LAPACK, with only a small loss in terms of reconstruction performance. In particular, our techniques yield a speedup of the ALM algorithm by more than 365 times compared to our reference C implementation, when tested with synthetic data. In order to demonstrate the effectiveness of our approach, we further optimize our algorithm for the foreground/background separation task considered in [3] and achieve real-time processing of 640×480 color video signals at 12 fps on an off-the-shelf CPU. For real-world video data, our methods result in 7 500 times lower computational complexity than our reference implementation.

Note that our approach differs fundamentally from the one proposed in [8]. There, a time-evolution model for the columns of the sparse and low-rank components of the matrix is considered. The algorithm proposed in [8] solves the decomposition problem for matrices drawn according to this model, whereas our methods can be applied to the general PCP problem. Furthermore, the adjective "real-time" in [8] has a different meaning than in this paper. In [8], it is used to highlight the fact that the proposed algorithm operates on-the-fly, i.e., on a column-by-column basis. Unlike in this paper, no actual measurements of the algorithm run-time are reported.

Notation: Matrices and vectors are denoted by uppercase and lowercase boldface letters, respectively. Throughout the paper, we shall deal exclusively with real-valued matrices. For a given matrix $\mathbf{A}$ with entries $A_{i,j}$, we define its ℓ_1-

norm to be $\|\mathbf{A}\|_1 \triangleq \sum_{i,j}|A_{i,j}|$, its nuclear norm to be $\|\mathbf{A}\|_* \triangleq \sum_i \sigma_i(\mathbf{A})$, where $\sigma_i(\mathbf{A})$ is the ith singular value of $\mathbf{A}$, and its Frobenius norm to be $\|\mathbf{A}\|_\mathrm{F} \triangleq \sqrt{\sum_{i,j}|A_{i,j}|^2}$. The inner product of two matrices $\mathbf{A}$ and $\mathbf{B}$ is defined as $\langle \mathbf{A},\mathbf{B}\rangle \triangleq \mathrm{tr}(\mathbf{A}^T\mathbf{B})$, where $\mathrm{tr}(\cdot)$ denotes the trace operator and $(\cdot)^T$ designates transposition. For a matrix $\mathbf{A} \in \mathbb{R}^{n_1\times n_2}$ we define the operator $\mathrm{vec}(\mathbf{A}) \triangleq [\mathbf{a}_1^T\ \mathbf{a}_2^T\ \cdots\ \mathbf{a}_{n_2}^T]^T$, where $\mathbf{a}_i$ is the ith column of $\mathbf{A}$. This means that $\mathrm{vec}(\mathbf{A})$ is the vertical stacking of all the columns of $\mathbf{A}$. We shall need the shrinkage operator $\mathcal{S}_\tau(\cdot)$, which maps $x \in \mathbb{R}$ to $\mathcal{S}_\tau(x) \triangleq \mathrm{sgn}(x)\max\{|x|-\tau,0\}$. Here, $\mathrm{sgn}(x)$ denotes the sign of $x\in\mathbb{R}$, which is taken to be zero if $x=0$. As in [3], the shrinkage operator is extended to matrices by applying it to each element. For a set of indices $\Omega \subset \mathbb{N}^2$, $\mathbf{A}_\Omega$ stands for the set of all values $A_{i,j}$ with $(i,j)\in\Omega$. Finally, $\mathcal{D}_\tau(\cdot)$ denotes the singular-value shrinkage operator, which is defined as $\mathcal{D}_\tau(\mathbf{A}) \triangleq \mathbf{U}\mathcal{S}_\tau(\mathbf{\Sigma})\mathbf{V}^T$, where $\mathbf{A}=\mathbf{U\Sigma V}^T$ is a singular-value decomposition (SVD) of the matrix $\mathbf{A}$.

II. Robust Principal Component Analysis

Let $\mathbf{M} = \widehat{\mathbf{L}}+\widehat{\mathbf{S}}$ be an $n_1\times n_2$ matrix which is the sum of a low-rank matrix $\widehat{\mathbf{L}}$ and a sparse matrix $\widehat{\mathbf{S}}$, i.e., a matrix with a small fraction of nonzero entries. Principal component pursuit (PCP) aims at recovering $\widehat{\mathbf{L}}$ and $\widehat{\mathbf{S}}$ by solving the following convex problem [3], [4]:

$$\text{(PCP)}\quad \begin{cases} \text{minimize} & \|\mathbf{L}\|_* + \lambda\|\mathbf{S}\|_1 \\ \text{subject to} & \mathbf{L}+\mathbf{S}=\mathbf{M}. \end{cases}$$

Here, the parameter $\lambda>0$ enables a tradeoff between rank and sparsity. Intuitively, PCP is related to the problem of decomposing $\mathbf{M}$ into $\widehat{\mathbf{L}}$ and $\widehat{\mathbf{S}}$, because the ℓ_1-norm enforces sparsity in $\mathbf{S}$ (a well-known property used extensively in the compressive-sensing literature [9]) and the nuclear norm enforces low rank in $\mathbf{L}$ (see [10] and references therein). Conditions under which PCP recovers $\widehat{\mathbf{L}}$ and $\widehat{\mathbf{S}}$ exactly have been reported recently in [3], [4].

One effective way to solve PCP for the case of large matrices is to use a standard augmented Lagrangian multiplier method [7]. This method consists of defining the augmented Lagrangian function

$$\ell(\mathbf{L},\mathbf{S},\mathbf{Y}) \triangleq \|\mathbf{L}\|_* + \lambda\|\mathbf{S}\|_1 + \langle \mathbf{Y},\mathbf{M}-\mathbf{L}-\mathbf{S}\rangle + \mu\|\mathbf{M}-\mathbf{L}-\mathbf{S}\|_\mathrm{F}^2 \quad (1)$$

and then minimizing it iteratively by setting

$$(\mathbf{L}^{(k)},\mathbf{S}^{(k)}) = \arg\min_{(\mathbf{L},\mathbf{S})} \ell(\mathbf{L},\mathbf{S},\mathbf{Y}^{(k)}) \quad (2)$$

and updating $\mathbf{Y}^{(k+1)} \leftarrow \mathbf{Y}^{(k)} + \mu(\mathbf{M}-\mathbf{L}^{(k)}-\mathbf{S}^{(k)})$. In (1), the matrix $\mathbf{Y}$ contains the Lagrangian multipliers, and $\mu>0$ is a tuning parameter that determines the convergence rate of the algorithm [2], [5]. As noted in [2], [3], [5], one can actually avoid performing the minimization (2) and use instead an alternating optimization approach (see, e.g., [11]), enabled by the fact that both $\arg\min_\mathbf{L} \ell(\mathbf{L},\mathbf{S},\mathbf{Y}^{(k)})$ and $\arg\min_\mathbf{S} \ell(\mathbf{L},\mathbf{S},\mathbf{Y}^{(k)})$ admit a simple closed-form expression

$$\arg\min_\mathbf{S} \ell(\mathbf{L},\mathbf{S},\mathbf{Y}) = \mathcal{S}_{\lambda\mu^{-1}}(\mathbf{M}-\mathbf{L}+\mu^{-1}\mathbf{Y}) \quad (3)$$

$$\arg\min_\mathbf{L} \ell(\mathbf{L},\mathbf{S},\mathbf{Y}) = \mathcal{D}_{\mu^{-1}}(\mathbf{M}-\mathbf{S}+\mu^{-1}\mathbf{Y}). \quad (4)$$

Hence, one can replace (2) with the two minimization problems (3) and (4), which results in the alternating-directions algorithm [5], [2, Alg. 5], and [3, Alg. 1] summarized in Algorithm 1. As suggested in [3], we shall use $\lambda = 1/\sqrt{\max\{n_1,n_2\}}$ and $\mu=(n_1n_2)/(4\|\mathbf{M}\|_1)$ in the remainder of the paper.

Algorithm 1 ALM using alternating directions [2], [3], [5]

```
1: input: $\mathbf{M} \in \mathbb{R}^{n_1\times n_2}$
2: initialize: $\mathbf{S}^{(0)} = \mathbf{Y}^{(0)} = \mathbf{0}$, $\lambda = 1/\sqrt{\max\{n_1,n_2\}}$,
   $\mu = (n_1n_2)/(4\|\mathbf{M}\|_1)$, $k=0$
3: while not converged do
4:     $\mathbf{L}^{(k+1)} \leftarrow \mathcal{D}_{\mu^{-1}}(\mathbf{M}-\mathbf{S}^{(k)}+\mu^{-1}\mathbf{Y}^{(k)})$
5:     $\mathbf{S}^{(k+1)} \leftarrow \mathcal{S}_{\lambda\mu^{-1}}(\mathbf{M}-\mathbf{L}^{(k+1)}+\mu^{-1}\mathbf{Y}^{(k)})$
6:     $\mathbf{Y}^{(k+1)} \leftarrow \mathbf{Y}^{(k)}+\mu(\mathbf{M}-\mathbf{L}^{(k+1)}-\mathbf{S}^{(k+1)})$
7: end while
8: output: $\mathbf{L}^{(k)},\mathbf{S}^{(k)}$
```

III. Methods for Complexity Reduction

The main goal of this work is to reduce the computational complexity of Algorithm 1, so as to be able to use it in applications requiring real-time processing. To achieve this, we describe several techniques that reduce the computational complexity at the cost of only a small penalty in terms of the recovery error.

A. Speeding up the SVD

The most computationally demanding step of Algorithm 1 is the application of the shrinkage operator $\mathcal{D}_{\mu^{-1}}(\cdot)$, which requires the computation of an SVD. In order to reduce the computational complexity of this step, it is important to realize that the shrinkage operator embedded in $\mathcal{D}_{\mu^{-1}}(\cdot)$ sets every singular value smaller than the threshold μ^{-1} to zero. Hence, only those singular values larger than μ^{-1} (and the corresponding singular vectors) need to be calculated. To take advantage of this property, we use the Power method [12] for calculating the SVD, since it enables us to compute the singular values in a sequential manner, and, hence, to stop the procedure as soon as a singular value smaller than μ^{-1} is found. The Power method, which is summarized below, typically performs better than the Hestenes/Nash [13] or Golub/Reinsch methods when dealing with large low-rank matrices [14].

Assume we wish to find the largest singular value and the associated left and right singular vectors of the matrix $\mathbf{A}\in\mathbb{R}^{n_1\times n_2}$. The Power method operates as follows: generate a non-zero vector $\mathbf{v}^{(0)}\in\mathbb{R}^{n_2}$, and repeat the following three steps for $k=1,2,\ldots$

1) $\mathbf{u}^{(k+1)} \leftarrow \mathbf{A}\mathbf{v}^{(k)}/\left\|\mathbf{A}\mathbf{v}^{(k)}\right\|_2$

2) $\mathbf{v}^{(k+1)} \leftarrow \mathbf{A}^T\mathbf{u}^{(k)} / \left\|\mathbf{A}^T\mathbf{u}^{(k)}\right\|_2$
3) $\sigma^{(k+1)} \leftarrow \left\|\mathbf{A}^T\mathbf{u}^{(k)}\right\|_2$.

until convergence is attained, that is until $\left\|\mathbf{v}^{(k+1)} - \mathbf{v}^{(k)}\right\|_2 < \delta_{\text{SVD}}\sqrt{n_2}$, for some small δ_{SVD}.

Here, $\sigma^{(k)}$ is an estimate of the largest singular value of $\mathbf{A}$ at iteration k, and $\mathbf{u}^{(k)}$ and $\mathbf{v}^{(k)}$ are the estimates of the corresponding left and right singular vectors, respectively. To compute the second largest singular value (and the associated left and right singular vectors), we just need to apply the Power method to $\mathbf{A} - \sigma\mathbf{u}\mathbf{v}^T$, where the triple $(\sigma, \mathbf{u}, \mathbf{v})$ is the output of the previous instance of the Power method. Obviously, this procedure can be generalized to any number of singular values. We stop the Power SVD algorithm when we find a singular value $\sigma_i < \mu^{-1}$ or when a certain predetermined number of singular values has been found. Note that if a good initial guess for $\mathbf{v}$ is available, the number of iterations required for convergence of the Power method decreases significantly [14].

B. Seeding the PCP Algorithm

In certain applications, the algorithm will operate on matrices consisting of blocks of contiguous frames; this is, for example, the case in foreground/background separation for video surveillance. More specifically, under the assumption of a stationary camera, the low-rank component is expected not to change significantly from one block to the next. Thus, we can use the low-rank part returned by the ALM algorithm from the previous block of data as a starting point for the next block. In this case, we begin running Algorithm 1 at Step 5 instead of at Step 4, after having set $\mathbf{L}^{(1)}$ to be equal to the low-rank matrix component that was output the last time the PCP algorithm was used.

C. Partitioning into Subproblems

In order to further reduce the computational complexity of Algorithm 1, we propose to partition the matrix $\mathbf{M}$ into P smaller submatrices. The hope is that, by combining the solutions of the P corresponding PCP subproblems, we can recover the solution of the original problem at lower computational complexity. The drawback of partitioning is that it is less likely that the recovery guarantees reported in [3] are satisfied for each individual subproblem; this eventually reduces the probability that the concatenated solution is correct.

Since the matrices we are interested in have considerably more rows than columns, we found that the best method for partitioning is to use a row-based scheme. Let Ω_i be chosen so that $\mathbf{M}_{\Omega_i}$ is the set of rows of $\mathbf{M}$ between $1 + (i-1)n_1/P$ and in_1/P. Then $\mathbf{M}_{\Omega_i}$ is the observation matrix for the ith subproblem. We finally note that partitioning enables us (i) to perform parallelization across multiple processing cores (e.g., CPUs or GPUs) and (ii) to handle larger dimensions by overcoming memory constraints. The corresponding optimal partition and parallelization scheme heavily depends on the target architecture. We present a specific example in Section V-C.

IV. Performance/Complexity Evaluation using Synthetic Data

As the basis of our performance and complexity comparisons, we consider a standard `C` implementation of Algorithm 1, which uses the LAPACK SVD routine `SGESVD` [15] returning *all* singular values. Whenever possible, the Intel math kernel library (MKL) is used to perform the calculations. The experiments presented in the following are carried out on a PC with an Intel i7 920 (Quad Core with Hyper-threading enabled) CPU and 4 GB of memory, clocked at 2.66 GHz and 1.066 GHz respectively. Unless stated otherwise, all libraries and code fragments were compiled so that they would use only one CPU core.

To characterize the performance gains resulting from our modifications on Algorithm 1, we use the following random test data. We generate low-rank matrices $\widehat{\mathbf{L}} = \mathbf{R}_1\mathbf{R}_2$ with $\mathbf{R}_1 \in \mathbb{R}^{n_1 \times r}$ and $\mathbf{R}_2 \in \mathbb{R}^{r \times n_2}$ independent matrices with i.i.d. Gaussian entries of mean zero and variance 1. The sparse matrix $\widehat{\mathbf{S}}$ is an all-zero matrix, except for $n_1n_2/20$ entries chosen uniformly at random. These entries are then assigned the values ± 1 with equal probability. The observed data is $\mathbf{M} = \widehat{\mathbf{S}} + \widehat{\mathbf{L}}$. We use $n_1 = 19\,200$, $n_2 = 200$ and $r = 10$, which models a block of 200 frames of video data, where each frame contains 160×120 pixels. We define the error in approximating $\widehat{\mathbf{L}}$ with $\mathbf{L}$ to be

$$\varepsilon(\widehat{\mathbf{L}}, \mathbf{L}) \triangleq \|\widehat{\mathbf{L}} - \mathbf{L}\|_{\text{F}} / \|\widehat{\mathbf{L}}\|_{\text{F}}$$

and similarly for $\widehat{\mathbf{S}}$ and $\mathbf{S}$. The total error is then

$$\varepsilon_T \triangleq \varepsilon(\widehat{\mathbf{L}}, \mathbf{L}) + \varepsilon(\widehat{\mathbf{S}}, \mathbf{S}).$$

We say that our algorithm has converged when it returns two matrices $\mathbf{S}$ and $\mathbf{L}$ satisfying $\varepsilon(\mathbf{M}, \mathbf{S} + \mathbf{L}) < \delta_{\text{PCA}}$ (thus we do not assume knowledge of the true solution). Note that this does not imply that $\varepsilon(\widehat{\mathbf{L}}, \mathbf{L}) < \delta_{\text{PCA}}$ or $\varepsilon(\widehat{\mathbf{S}}, \mathbf{S}) < \delta_{\text{PCA}}$. We use the thresholds $\delta_{\text{SVD}} = \delta_{\text{PCA}} = 10^{-4}$.

A. Impact of the Power SVD Algorithm

The use of the Power method instead of the LAPACK SVD routine (which, in contrast to the Power method, finds all the singular values and vectors) results in 4.32× lower runtime. If we stop computing singular values as soon as we find one below the threshold, we gain a further 2.02× speedup. If we assume that the rank of $\widehat{\mathbf{L}}$ is known and we terminate the Power SVD algorithm when we have found more than $\text{rank}(\widehat{\mathbf{L}})$ singular values, we get an additional 17.35× speedup.

B. Impact of Seeding the SVD and PCP Algorithm

Using the seeding techniques of [13] for the Power SVD algorithm yields an additional 1.73× speedup. For the considered synthetic data, the low rank components are independent from one problem to the next, and therefore, no speedup can be obtained by seeding the ALM algorithm.

C. Impact of Partitioning

By subdividing $\mathbf{M}$ into 8 sub-matrices and using a row-based partitioning scheme, we achieve a 1.4× lower run-time.

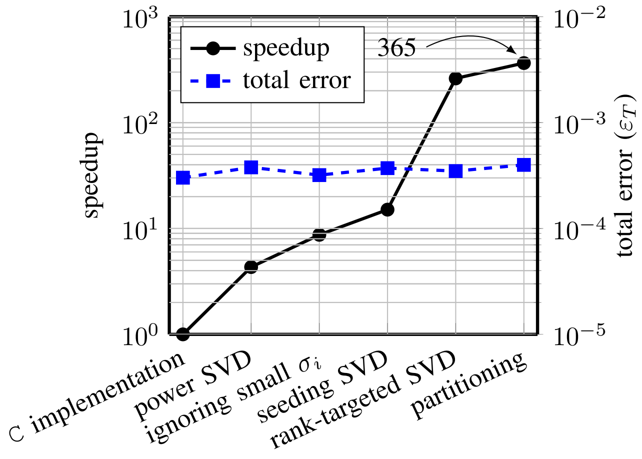

Figure 1. Speedup and recovery performance breakdown for synthetic data.

D. Summary of the Achieved Complexity and Performance

Fig. 1 shows the individual impact of the techniques presented in Section IV-A to IV-C. The combination of all methods results in an overall complexity reduction by a factor of 365 over the original C implementation using the standard LAPACK SVD algorithm. Furthermore, as shown in Fig. 1, the impact of the proposed algorithmic modifications to the recovery error turns out to be marginal, which proves the effectiveness of our methods.

V. APPLICATION EXAMPLE: REAL-TIME FOREGROUND/BACKGROUND SEPARATION

In order to demonstrate the effectiveness of the methods proposed in Section III, we consider the real-time separation of a video stream into foreground and background components.

A. Formulation as an RPCA Problem

The separation of a video stream into foreground and background can be set up as an RPCA problem as follows. Let $\{\mathbf{F}_i\}$ be a collection of F consecutive video frames of dimension $n_1 \times n_2$ and form the vectorized version $\mathbf{f}_i = \text{vec}(\mathbf{F}_i) \in \mathbb{R}^{n_1 n_2}$ of each frame $\mathbf{F}_i$. Set $\mathbf{M} = [\mathbf{f}_1 \cdots \mathbf{f}_F]$, which is then a matrix of size $n_1 n_2 \times F$. Following the reasoning in [3], the foreground consists of the moving objects and can thus be modeled as a sparse matrix $\mathbf{S}$, and the background consists of stationary objects and is thus low rank. Note that the foreground data is not actually directly contained in the matrix $\widehat{\mathbf{S}}$, which is the difference between the observed frames and the low-rank component. The foreground image is rather given by $\mathbf{M}_\Omega$, where Ω is the location of the non-zero components of $\widehat{\mathbf{S}}$.

We next apply our ALM algorithm to a test video-stream to show how well our techniques work with real world data. These simulations are performed with 8-bit greyscale video frames of dimension 160×120 with an incoming frame rate of 10 fps. We process 25 frames in one block, i.e., $F = 25$, which means that our implementation must be able to process each block in less than 2.5 seconds, to be able to run in real-time. The speedup gains using the techniques discussed earlier are illustrated in Fig. 2.

B. Impact of the Power SVD Algorithm

The use of the Power SVD algorithm in place of the LAPACK SVD results in an 83× speedup, even when we calculate all the singular values. If we abort the Power SVD algorithm when we find a singular value less than μ^{-1}, we further decrease the runtime by a factor of 5.2. Using the previously calculated singular vectors to form estimates of the new ones, we further decrease the complexity by a factor of 1.32. The combined effect of all of these SVD optimizations nets us a speed improvement by a factor of 578.

C. Parallelization and Partitioning for a Multi-Core CPU

We now explore a combination of partitioning and parallelization to further reduce complexity. As the partitioning is, in part, motivated by a desire to make it easier to parallelize, we compare the following options for parallelization: (i) solve each subproblem in parallel, (ii) implement the matrix-vector arithmetic in parallel, or (iii) a combination of the previous two. For the target CPU (having four cores and using hyper-threading), the best results were achieved by running two problems in parallel and letting the LAPACK libraries run on two cores each; this approach resulted in a 2.35× speedup.

D. Final Tweaking of the Parameters

In order to further speed up the PCP algorithm, we optimized some of the thresholds. Since we do not know the ground truth, we compare the results of the original LAPACK implementation with the optimized version, to ensure that we still have a good solution. Let $\mathbf{S}_0$ and $\mathbf{L}_0$ be the solutions returned by the LAPACK implementation and let $\mathbf{S}$ and $\mathbf{L}$ be the solutions returned by the modified algorithm, then we take as error

$$\varepsilon_L = \varepsilon(\mathbf{S}_0, \mathbf{S}) + \varepsilon(\mathbf{L}_0, \mathbf{L})\,.$$

We found that δ_{PCA} and δ_{SVD} could be increased while maintaining an error satisfying $\varepsilon_L < 10^{-3}$, which resulted in a speedup by a factor of 1.41. Instead of a numerical constraint, if we merely require that there is "little noticeable visual difference" between the original and new solutions, we can further decrease the complexity of the algorithm by a factor of 3.15. We do this in part by limiting the number of iterations that the ALM and Power SVD algorithms can run for. This early-termination scheme has the added advantage that it caps the amount of time that the ALM algorithm can operate, so we can force the program to process data in real-time.

E. Final Results

Fig. 2 illustrates the speedups achieved for real-time separation of a video stream. In summary, the final implementation is 7 544× faster than the naive C implementation using the LAPACK SVD routine. Our final implementation took, on average, 0.17 seconds to process 25 frames of data. Thus, the resulting algorithm is fast enough to allow for the processing of a

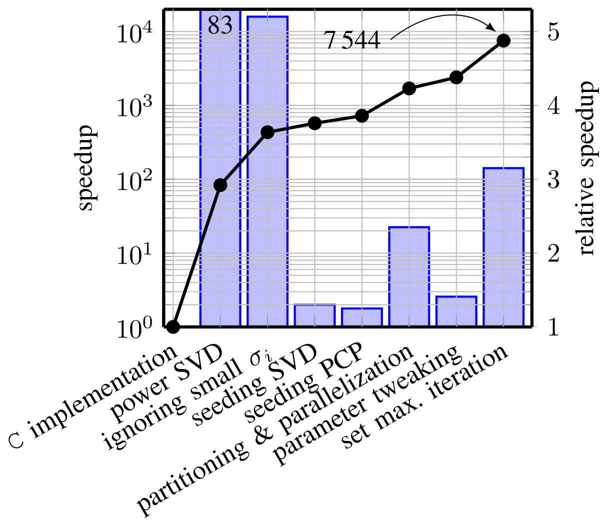

Figure 2. Individual speedups for video data.

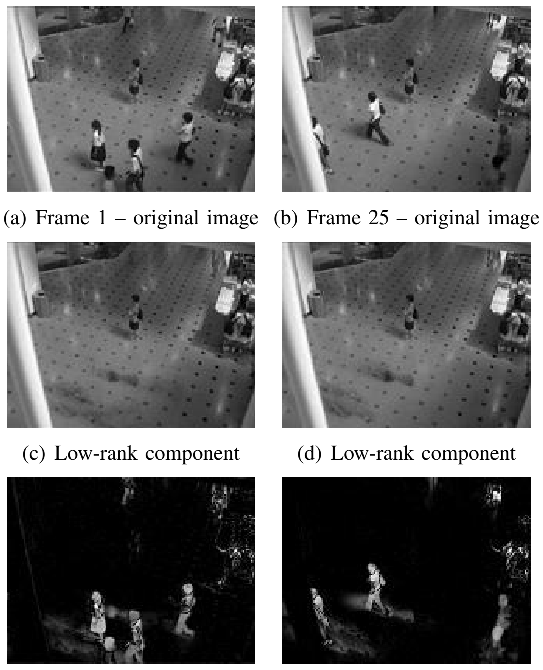

Figure 3. Two frames of an analyzed video sequence taken in a shopping mall [16]. The sparse components show the moving people and their shadows. The low-rank components show the background and the single stationary person.

video stream in real-time, even when the frame resolution is increased up to 640×480 pixels. An example of the separation result is shown in Fig. 3.

F. Handling Color Videos

So far we have only dealt with greyscale images. We use a simple method for processing a color video sequence. Let $\mathbf{M}$ be a grayscale version of the video sequence and let $\mathbf{M}^{(\ell)}$ for $\ell = 1, 2, 3$ be the three observed color components. Run the algorithm on $\mathbf{M}$ to get the low-rank matrix $\mathbf{L}$ and the sparse matrix $\mathbf{S}$ and let Ω be the locations of the non-zero components of $\mathbf{S}$. We use Ω to specify the locations of the color channels $\mathbf{M}^{(\ell)}$, so that the color version of the foreground is given by $\mathbf{M}^{(\ell)}_{\Omega}$. This method works since we are not interested in the sparse components per se, but rather only their locations. Unfortunately, we cannot use this approach to directly get the low-rank color approximation. We are, however, able to greatly simplify the computations by taking advantage of the fact that the locations of the non-zero components are known, so the problem reduces to the classical matrix completion problem [10].

VI. Conclusion

In this paper, we detailed a variety of methods that significantly reduce the computational complexity of principal component pursuit for robust principal component analysis. Furthermore, we demonstrated that the the PCP algorithm of [3] is in fact suitable for real-time foreground/background separation for video-surveillance application using off-the-shelf hardware. Our linux software for real-time principal component pursuit is available at http://www.nari.ee.ethz.ch/commth/research/downloads/.

References

[1] I. T. Jolliffe, *Principal Component Analysis*, 2nd ed., ser. Springer Series in Statistics. New York, NY, U.S.A.: Springer, 2002.

[2] Z. Lin, M. Chen, L. Wu, and Y. Ma, "The augmented Lagrange multiplier method for exact recovery of corrupted low-rank matrices," Oct. 2009. [Online]. Available: http://arxiv.org/abs/1009.5055

[3] E. J. Candès, X. Li, Y. Ma, and J. Wright, "Robust principal component analysis?" Jan 2009. [Online]. Available: http://arxiv.org/abs/0912.3599v1

[4] V. Chandrasekaran, S. Sanghavi, P. A. Parrilo, and A. S. Willsky, "Rank-sparsity incoherence for matrix decomposition," *SIAM J. Opt.*, vol. 21, no. 2, pp. 572–596, 2011.

[5] X. Yuan and J. Yang, "Sparse and low-rank matrix decomposition via alternating direction methods," *preprint*, 2009.

[6] M. Grant and S. Boyd, "CVX: Matlab software for disciplined convex programming." [Online]. Available: http://cvxr.com/cvx/

[7] D. P. Bertsekas, *Constrained Optimisation and Lagrange Multiplier Methods*. Belmont, MA, U.S.A.: Athena Scientific, 1982.

[8] C. Qiu and N. Vaswani, "Real-time robust principal components' pursuit," in *Proc. Allerton Conf. Commun., Contr., Comput.*, Monticello, IL, U.S.A., Oct. 2010, pp. 591–598.

[9] A. Bruckstein, D. Donoho, and M. Elad, "From sparse solutions of systems of equations to sparse modeling of signals and images," *SIAM Review*, vol. 51, no. 1, pp. 34–81, 2009.

[10] E. J. Candès and B. Recht, "Exact matrix completion via convex optimization," *Found. Comput. Math.*, vol. 9, no. 6, pp. 717–772, Apr. 2009.

[11] J. C. Bezdek and R. J. Hathaway, "Some notes on alternating optimization," *Lecture Notes in Computer Science*, vol. 2275, pp. 187–195, 2002.

[12] S. Shlien, "A method for computing the partial singular value decomposition," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. PAMI-4, no. 6, pp. 671 – 676, Jun. 1982.

[13] J. C. Nash, "A one-sided transformation method for the singular value decomposition and algebraic eigenproblem," *Computer Journal*, vol. 18, no. 1, pp. 74–76, Jan. 1975.

[14] J. C. Nash and S. Shlien, "Simple algorithms for the partial singular value decomposition," *Computer Journal*, vol. 30, no. 3, pp. 268–275, Jan. 1987.

[15] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, 3rd ed. Philadelphia, PA, U.S.A: Society for Industrial and Applied Mathematics, 1999.

[16] L. Li, W. Huang, I. Y.-H. Gu, and Q. Tian, "Statistical modeling of complex backgrounds for foreground object detection," *IEEE Trans. Image Process*, vol. 13, no. 11, pp. 1459 –1472, Nov. 2004.