Master Thesis

Iterative methods for matrix factorization with missing data

Author(s):
Daskalov, Boris Nikolaev

Publication Date:
2011

Permanent Link:
https://doi.org/10.3929/ethz-a-006607487

Rights / License:
In Copyright - Non-Commercial Use Permitted
Abstract

This thesis discusses the problem of reconstructing matrices with missing data by utilizing a low-rank constraint. This is an important subproblem in structure-from-motion (SFM) tasks in computer vision. It also has applications in other fields such as collaborative filtering. This work concentrates on several algorithms that are commonly used to solve such problems: Alternating Least Squares (ALS), Levenberg-Marquardt (LM) and Wiberg’s algorithm (WA). These algorithms were compared on synthetic low-rank matrices. Also they were generalized to operate on affine structure-from-motion (SFM) problems and tested on synthetic and real-world SFM data. The results showed that WA clearly outperforms other methods by being able to reconstruct the same matrices using less observations. Also for the same number of observations WA converges to the global minimum more often and in less iterations than LM and ALS.

Two improvements in the WA are proposed in this thesis. The first one is a way to apply iterative methods to solve for the step direction. Our approach speeds up the most computationally intensive part of the algorithm by exploiting the sparsity and the algebraic properties of the problem. The per-iteration computational complexity of the proposed variant is similar or lower than the LM algorithm that is commonly used for bundle adjustment. It converges faster than LM and finds the global minimum more often when started from a random point. This makes WA practical to apply to large SFM problems. The other proposed improvement is adding soft orthogonality constraints to the optimization problem in a way that preserves the least-squares form of the cost function. In our experiments this reduced the number of runs that get stuck in a local minimum and further increased the convergence speed of Wiberg’s algorithm.
1 Introduction

This thesis discusses the problem of low-rank matrix factorization in the presence of missing data. This problem appears in the context of computer vision as the central step in structure-from-motion (SFM) tasks. Another application includes collaborative filtering and illumination based reconstruction. The problem is also related to principal component analysis with missing data.

Structure from motion is the problem of simultaneously deriving the 3D structure of the scene and the motion of the camera over multiple views. Usually in such tasks the input consists of the trajectories of multiple scene features as observed in all or part of the views.

In [11] Tomasi and Kanade discussed a model in which the feature trajectories are fully observed and orthographic camera model is assumed. They showed that under those conditions the matrix of feature trajectories has rank three because it can be decomposed as the product two factors. One of them contains the stacked camera projection matrices and the other - the stacked feature point coordinates. The best, in least squares sense, rank-three decomposition of a matrix can be easily reconstructed using singular value decomposition. Then the factorization can be used to recover the structure and motion parameters. The Tomasi-Kanade factorization model has been generalized to affine (Poelman and Kanade[8]) and projective cameras(Sturm and Triggs[10]).

The main problem in factorization based approaches to SFM is handling missing data. Feature trajectories are usually constructed using tracker algorithms which are prone to errors. Even if the tracker is perfect a feature can be occluded in some of the views which makes it impossible to track and leads to incomplete trajectories. Usually similar sets of features are observed in consecutive views and this causes SFM problems to have a specific observation structure where most of the observed entries are located close to the main diagonal of the matrix. Because of the missing data problem, a popular approach to SFM problems is to create partial reconstructions from consecutive two- or three-view problems and "stitch" them together. This "stitched" solution is later refined to a globally optimal solution through the so called bundle adjustment process which usually uses the Levenberg-Marquardt (LM) algorithm. This standard approach to SFM is analogous to running an iterative method for matrix factorization with a good initial guess. In this work we will assume that such initial guess is not available and explore when factorization algorithms can recover the underlying structure without it. We will compare Levenberg-Marquardt, which is a generic algorithm for least-squares problems, to Alternating Least Squares and Wiberg’s algorithm, which are specialized methods for low-rank matrix factorization. The main goal of this thesis is to explore the effect of missing data on the algorithms that are commonly used to solve low-rank factorization problems. The main question is how does their performance depend on the quantity of observed matrix entries. Another question that we tried to address is what is the influence of the diagonal structure that is typical for SFM problems on the performance of the algorithms.

The structure of this thesis is the following:

- Sec. 2 describes the notation that is used in the rest of the thesis.
Sec. 3 gives a short overview of the methods that are commonly used to solve linear and non-linear least-squares problems.

Sec. 4 explains how matrix factorization is defined as least-squares minimization problem and shows some of its properties.

Sec. 5 defines the algorithms that were compared in the experiments.

Sec. 6 describes the experiments on synthetic low-rank matrices and discusses the results from them.

Sec. 7 explains the connection between low-rank matrix factorization and SFM problems and shows how the algorithms presented in Sec. 5 can be generalized to operate on affine SFM data.

Sec. 8 presents experiments on synthetic and real-world SFM data and discusses the results.
2 Notation

In this work vectors will be noted with bold lower-case letters and matrices with bold upper-case letters. The sizes of matrices will be given as a subscript sometimes (e.g. \( A_{m \times n} \)). The identity matrix of size \( n \times n \) will be denoted as \( I_n \). The size may be omitted in some cases.

\[
P_A = A(A^T A)^{-1} A^T \text{ is a projection to the column space of the matrix } A \text{ and } Q_A = I - P_A \text{ is a projection to the column space orthogonal to the column space of } A. \quad \text{An orthogonal basis of the column space of } A \text{ will be denoted as } O_A. \]

\[
\| A \|_F \quad \text{denotes the Frobenius norm of the matrix } A \text{ and } \| x \|_2 \quad \text{- the 2-norm of the vector } x. \quad \text{\( \mathbf{0} \) stands for the zero vector and } \mathbf{1} \text{ for the vector with all elements equal to one. All vectors are column-vectors.}
\]

2.1 Vectorization Operator and Kronecker Product

In this work we will need to treat matrices as elements of a linear space and will use the vectorization operator \( \text{vec } [A] \). This operator constructs a vector from a matrix by concatenating all the columns as follows:

\[
\text{vec} \begin{bmatrix}
  a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\
  a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\
  \vdots & \vdots & \ddots & \vdots \\
  a_{m,1} & a_{m,2} & \cdots & a_{m,n}
\end{bmatrix} =
\begin{bmatrix}
  a_{1,1} \\
  \vdots \\
  a_{m,1} \\
  a_{1,2} \\
  \vdots \\
  a_{m,n}
\end{bmatrix}
\]

The Kronecker product denoted by \( \otimes \) is a matrix-matrix operation defined as:

\[
A_{m \times n} \otimes B_{p \times q} =
\begin{bmatrix}
  a_{11}B & \cdots & a_{1n}B \\
  \vdots & \ddots & \vdots \\
  a_{m1}B & \cdots & a_{mn}B
\end{bmatrix}_{mp \times nq}
\]

There is an important connection between the vectorization operator and the Kronecker product:

\[
\text{vec } [AXB] = (B^T \otimes A) \text{ vec } [X]
\]

This property allows us to rearrange a matrix-matrix product as a matrix vector product as follows:

\[
\text{vec } [AX] = \text{vec } [AXI] = (I \otimes A) \text{ vec } [X]
\]

\[
\text{vec } [XA] = \text{vec } [IXA] = (A^T \otimes I) \text{ vec } [X]
\]

This is especially useful when derivatives of matrices have to be computed. Using vectorization one can calculate the derivative of a matrix with respect to another matrix by reshaping both matrices as vectors and taking the Jacobian. Some other properties of the Kronecker product that will be used are:
• Bi-linearity:
\[(A + B) \otimes C = (A \otimes C) + (B \otimes C)\]  
\[A \otimes (B + C) = (A \otimes B) + (A \otimes C)\]  
\[\lambda(A \otimes B) = (\lambda A) \otimes B = A \otimes (\lambda B)\]  

• Mixed-product property:
\[(A \otimes B)(C \otimes D) = AC \otimes BD\]  
when the products \(AC\) and \(BD\) can be formed.

• Transposition:
\[(A \otimes B)^T = A^T \otimes B^T\]  

• Inversion:
\[(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}\]  

Another matrix-matrix operation that will be used is the element-wise or Hadamard product denoted by \(\otimes\). It is defined as:

\[\mathbf{A}_{m \times n} \otimes \mathbf{B}_{m \times n} = \begin{pmatrix} a_{11} b_{11} & \cdots & a_{1n} b_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} b_{mn} & \cdots & a_{mn} b_{mn} \end{pmatrix}_{m \times n}\]
3 Solving Least-Squares Problems

Linear and non-linear least-squares (LS) problems are central building blocks for the methods that will be presented. This section is a brief introduction to LS problems and algorithms for solving them. If the reader is familiar with that material it can be safely skipped as it does not include anything specific to matrix factorization. More comprehensive discussion of the material in this section can be found in [6].

3.1 Definition of a Least-Squares Problem

A least-squares problem is defined as:

\[
\min_x F(x)
\]

where

\[
F(x) = \sum_{i=1}^{m} r_i^2(x)
\]

(13)

Such problems often arise when fitting a model to a set of observations in a way that minimizes the sum of squared errors. Typically each function \( r_i(x) \) represents the difference between predicted and observed values for one data point. The vector \( r(x) = [r_1(x) \ r_2(x) \ldots \ r_m(x)] \) is called residual vector. A least squares problem can be formulated in terms of the residual vector as follows:

\[
\min_x F(x)
\]

where

\[
F(x) = \|r(x)\|^2 = r^T(x)r(x)
\]

(14)

A necessary but not sufficient condition for \( x \) to be a solution of a LS problem is the following:

\[
0 = \frac{\partial F(x)}{\partial x} = 2 \frac{\partial r(x)}{\partial x}^T r(x)
\]

(15)

We will note the Jacobian \( \frac{\partial r(x)}{\partial x} \) with \( J \) and write to optimality condition as:

\[
J^T r = 0
\]

(16)

3.2 Solving Linear Least-Squares Problems

In cases where \( r(x) \) is a linear function:

\[
r(x) = Ax - b
\]

(17)

the Jacobian is:

\[
J = \frac{\partial r}{\partial x} = A
\]

(18)

and the corresponding optimality condition is:

\[
A^T (Ax - b) = \bar{0}
\]

(19)
\[ A^T A x = A^T b \] (20)

This system can be solved to \( x = (A^T A)^{-1} A^T b \) provided that \( A \) has a full rank. If \( A \) is rank deficient there is a linear subspace of equivalent solutions that have the same cost function value.

In practice inverting the matrix \( A^T A \) or solving the normal equations (20) is computationally expensive and numerically unstable. This is why LLS problems are usually solved using QR decomposition. The complexity of solving a dense LLS problem with \( m \) equations and \( n \) variables using QR is \( O(mn^2) \).

In cases where the matrix \( A \) is sparse the speed can be improved by using iterative methods like CGLS or LSQR. Such methods need only subroutines that multiply an arbitrary vector by \( A \) or \( A^T \). These multiplications need time that is proportional to the number of non-zero elements of \( A \). The number of matrix-vector multiplications needed in the worst case is \( O(n) \). The complexity of solving a sparse LLS problem is \( O(ln) \) where \( l \) is the number of non-zero entries in \( A \). It is important to note that \( O(n) \) multiplications is a very pessimistic bound and these methods usually reach a satisfactory solution much faster.

### 3.3 Solving Non-Linear Least-Squares Problems

Solvers for non-linear least-squares problems are usually based on the Gauss-Newton algorithm. This method needs a starting point \( x_0 \). Then it creates a local linear approximation of the residual around \( r(x_0) \):

\[
 r(x_0 + \Delta x) \approx r(x_0) + J(x_0) \Delta x \] (21)

The next step is to solve the linear least squares problem:

\[
 \Delta x = \arg\min_{\Delta x} \| r(x_0) + J(x_0) \Delta x \|_2^2 \] (22)

The solution \( \Delta x \) is a descent direction for \( F(x) \). After computing it the algorithm performs a step in this direction. It is important to control the step size since this guarantees that the function value will decrease and that the algorithm will converge. Different Gauss-Newton based methods differ mostly in their approach to controlling the step size. There are three methods to do that:

- **Line search**
  This method solves a one dimensional optimization problem to determine a good step size. It finds a local minimum on the line defined by the current point and the descent direction. Generally this is done using one dimensional black-box optimization like golden-ratio search, but in some cases it is possible to perform the line search analytically.

- **Levenberg-Marquardt**
  This approach extends the Gauss-Newton algorithm by adding a step-size penalty when solving for the step direction in (22). The regularization parameter is changed dynamically. Effectively this leads to interpolation of the step direction between the GN direction and the direction of steepest descent.
• Trust regions
This is another approach that interpolates between the GN direction and
the direction of steepest descent. It does not regularize the optimization
problem \(^{22}\) but defines a region in which it is trusted to be a good ap-
proximation. The size of the region is dynamically modified. This method
is very similar to Levenberg-Marquardt although there is no analytical
connection between them.
4 Low-Rank Matrix Factorization as a Non-Linear Least Squares Problem

The main subject of this work is the problem of recovering a matrix $X_{m \times n}$ with a given low rank $k \ll m < n$. The low rank constraint implies that $X$ can be represented as a product of two matrices $U_{m \times k}$ and $V_{k \times n}$ such that:

$$X_{m \times n} = U_{m \times k} V_{k \times n}$$ (23)

If we note the columns of $V$ with $\{v_j\}_{j=1..n}$ then the $j$-th column of $X$ is equal to $Uv_j$. Each column of $X$ is a linear combination of the columns of $U$ with coefficients $v_j$. This means that $U$ is a basis of the column space of $X$. Similarly $V$ is a basis of the row space of $X$.

Of course this representation is not unique. There are multiple bases of the row and column spaces of $X$. Actually for each invertible matrix $Q_{k \times k}$ one can create an equivalent factorization:

$$X_{m \times n} = U'_{m \times k} V'_{k \times n}$$

where

$$U'_{m \times k} = U_{m \times k} Q_{k \times k}$$
$$V'_{k \times n} = Q^{-1}_{k \times k} V_{k \times n}$$ (24)

Because of this ambiguity, the factorization has $mk + nk - k^2$ degrees of freedom. For small values of $k$ this is much smaller than the number of degrees of freedom of $X$ which is $mn$. This makes it possible to recover the full matrix $X$ by observing only a subset of its elements.

4.1 Vectorized formulation

The set of matrices with fixed size is a linear space. When fitting our model to observations it is useful to treat $X$, $U$ or $V$ as vectors of such linear space. It is convenient to have a way of writing the problem in which their elements are arranged in vectors. This can be achieved easily using the Kronecker product property of the vectorization operator.

$$\text{vec}[X] = \text{vec}[UV]$$
$$= (V^T \otimes I_m) \text{vec}[V]$$
$$= (I_n \otimes U) \text{vec}[U]$$ (25)

This is an equivalent representation of a low rank matrix factorization which is much more convenient to work with when fitting the model to data. In this thesis both representations will be used depending on which is more suitable for the particular situation. The matrix formulation is easier to interpret, whereas the vectorized formulation is useful since most optimization algorithms expect the parameters to be arranged in a vector and need to compute derivatives.
4.2 Missing data and noisy observations

We will assume that there is a ground truth matrix $X_{m \times n}$ which we are trying to reconstruct and which has a known low rank $k$. Our input data consists of the matrices $X_{m \times n}$ and $M_{m \times n}$ such that:

$$X = M \odot (X + \text{noise})$$  \hspace{1cm} (26)

$M$ is a boolean mask over that selects an observed subset of the elements. This means that if $m_{ij} = 1$ then the corresponding element $x_{ij}$ is observed and $x_{ij} = x_{ij} + \eta_{ij}$ where $\eta_{ij}$ is a noise term. Since $X_{m \times n}$ has a fixed low rank $k$, it can be factorized into $U_{m \times k}$ and $V_{k \times n}$:

$$X = UV$$
$$X = M \odot (UV + \text{noise})$$

$x_{ij}$ and $X_{m \times n}$ are related by the following equations:

$$x = S \text{vec} [X]$$
$$\text{vec} [X] = S^T x$$  \hspace{1cm} (28)

$S$ is a matrix that discards the unnecessary elements of $\text{vec} [X]$ (those that are removed by the zero elements of $M$). It is also possible to do the reverse operation in which we fill in the missing elements by zeros by multiplying with the transpose of $S$. $S$ is such that its rows are a subset of the rows of the identity matrix and it is functionally dependent on $M$.

For convenience we will use the following notation:

$$F = S(V^T \otimes I)$$
$$G = S(I_n \otimes U)$$
$$u = \text{vec} [U]$$
$$v = \text{vec} [V]$$
$$x = S \text{vec} [X]$$

This way the model is formulated as:

$$x = Fu = Gv$$  \hspace{1cm} (30)

4.3 Error function

For the sake of simplicity we assume that the noise is i.i.d. Gaussian and it’s magnitude is small. In such models the maximum likelihood solution is given by the global minimum of the least squares error function:
\[ E(U, V) = \| M \odot (UV - \mathbf{X}) \|_p^2 \] \hspace{1cm} (31)

We can also define the error function in the terms of the residual matrix:

\[ R = M \odot (UV - \mathbf{X}) \] \hspace{1cm} (32)

\[ E(U, V) = \| R \|_p^2 \] \hspace{1cm} (33)

A corresponding definition for the vectorized model is:

\[ r = F\mathbf{u} - \mathbf{x} = G\mathbf{v} - \mathbf{x} \] \hspace{1cm} (34)

\[ e(\mathbf{u}, \mathbf{v}) = \| r \|_2^2 = r^T r \] \hspace{1cm} (35)

A summary of the values involved in the definition of the error function and the connections between the matrix and vector formulations can be found in Tab. 4.4.

It is important to note that minimizing the \( l_2 \) norm of the error is not the optimal choice for all applications. It is particularly not suitable for cases where there are outliers in the training examples. Robust cost functions are often used in practice in order to reduce the effect of outliers on the final solution. An example of a matrix factorization method that minimizes \( l_1 \) norm in order to be robust to outliers can be found in [2].

### 4.4 Bi-linearity

An important feature of the model is that it is linear in both \( U \) and \( V \). This means that one can find an optimal closed form solution for \( U \) given \( V \). More precisely we have a linear least-squares problem in \( \mathbf{u} \) or \( \mathbf{v} \) and the closed form solution is given by multiplying with the pseudo-inverse of \( F \) or \( G \).

\[ \text{vec} \left[ \hat{U} \right] = \hat{\mathbf{u}} = (F^T F)^{-1} F^T \mathbf{x} \] \hspace{1cm} (36)

\[ \text{vec} \left[ \hat{V} \right] = \hat{\mathbf{v}} = (G^T G)^{-1} G^T \mathbf{x} \] \hspace{1cm} (37)
<table>
<thead>
<tr>
<th>Matrix formulation</th>
<th>Vectorized formulation</th>
<th>Connection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows</td>
<td>m</td>
<td></td>
</tr>
<tr>
<td>Columns</td>
<td>n</td>
<td></td>
</tr>
<tr>
<td>Rank</td>
<td>k</td>
<td></td>
</tr>
<tr>
<td>Observed elements</td>
<td>l</td>
<td></td>
</tr>
<tr>
<td>Ground truth</td>
<td>X</td>
<td>x</td>
</tr>
<tr>
<td>Low-rank model</td>
<td>U, V</td>
<td>u, v</td>
</tr>
<tr>
<td>Observations</td>
<td>( \overline{X} )</td>
<td>( \overline{x} )</td>
</tr>
<tr>
<td>Mask / Projection</td>
<td>M</td>
<td>S</td>
</tr>
<tr>
<td>Residual</td>
<td>R = M ⊙ (UV - ( \overline{X} ))</td>
<td>r = Fu - ( \overline{x} ) = Gv - ( \overline{x} )</td>
</tr>
<tr>
<td>Error function</td>
<td>( E(U, V) = |R|^2 )</td>
<td>( e(u, v) = |r|^2 )</td>
</tr>
</tbody>
</table>

Table 1: Notation reference.
5 Algorithms

Sec. 4 defined the task of fitting a low-rank factorization to a partially observed matrix as a non-linear least squares problem. Such problems are most commonly solved by algorithms that are based on the Gauss-Newton method. Our particular problem has a specific structure that can be exploited in order to speed up such generic methods. In this section we will describe how this can be accomplished. Also we will look into some methods that are specific to the low-rank factorization task like Alternating Least Squares and Wiberg’s algorithm. In the end we will give a quick overview of some other methods that are used mostly for collaborative filtering problems and as our experiments have shown do not perform that well for matrix factorizations that represents structure from motion tasks.

5.1 Gauss-Newton based algorithms

Gauss-Newton is a generic method for non-linear least squares problem. In each iteration it constructs a linear approximation of the residual around the current value of the parameters $x$:

$$r(x + \Delta x) \approx r(x) + \frac{\partial r(x)}{\partial x} \Delta x \quad (38)$$

Then it solves the linear least squares problem:

$$\arg\min_{\Delta x} \left\| r(x) + \frac{\partial r(x)}{\partial x} \Delta x \right\|^2_2 \quad (39)$$

Then the algorithm makes a step in the direction defined by $\Delta x$, which is guaranteed to be a descent direction. There are various ways to select the step size in a way that ensures that the value of the cost function will be lower in the next iteration.

5.1.1 Solving for the step direction

As defined in (34) the residual of the matrix factorization problem is:

$$r = Fu - x = Gv - x \quad (40)$$

Since $r$ is linear in $u$ and $v$ separately, the Jacobians with respect to them are obviously:

$$J_u = \frac{\partial r}{\partial u} = F \quad (41)$$

$$J_v = \frac{\partial r}{\partial v} = G \quad (42)$$

In order to apply the GN algorithm, one must combine the parameters in a single vector. We chose to stack $u$ and $v$ vertically as $[u \ v]$. This way the Jacobian
with respect to the full parameter vector is the horizontal concatenation of $J_u$ and $J_v$:

$$J = \begin{bmatrix} J_u & J_v \end{bmatrix} = \begin{bmatrix} F & G \end{bmatrix} = \begin{bmatrix} S(V^T \otimes I_m) & S(I_n \otimes U) \end{bmatrix} = S \begin{bmatrix} V^T \otimes I_m & I_n \otimes U \end{bmatrix}$$ (43)

The next step is to solve the overdetermined linear system $J \begin{bmatrix} \Delta u \\ \Delta v \end{bmatrix} = -r$ in least squares sense. It is important to take into account the fact that $J = [F \ G]$ and $F$ and $G$ have exactly $lk$ non-zero elements each. So $J$ has only $2lk$ non-zero elements out of $l(m+n)k$ and using a sparse least squares solver for this step leads to a much more efficient implementation. There are two types of such solvers: direct and iterative. Direct algorithms for solving linear systems operate on the matrix that defines the system. On the other hand iterative solvers only need an user-supplied procedure that multiplies an arbitrary vector by $J$ or its transpose. Because of the specific structure of the Jacobian these multiplications can be done without explicitly constructing $J$.

When doing the multiplication $Jy$, one can interpret $y$ as a change in the parameters:

$$y = \begin{bmatrix} \Delta u \\ \Delta v \end{bmatrix}$$

$$Jy = S \begin{bmatrix} V^T \otimes I_m & I_n \otimes U \end{bmatrix} \begin{bmatrix} \Delta u \\ \Delta v \end{bmatrix} = S((V^T \otimes I_m)\Delta u + (I_n \otimes U)\Delta v) = S(\text{vec}[\Delta U V] + \text{vec}[U \Delta V])$$ (44)

This can be computed in $O(lk)$ time if one does not compute the whole matrices $\Delta U V$ and $U \Delta V$ but only the elements that are observed (the ones that are not masked by $M$). This is actually the same complexity as multiplying by a sparse version of $J$ but it does not require constructing it explicitly.

Similarly, when computing $J^Ty$, one can interpret $y$ as a change in the
residual:

\[ y = \Delta r \]

\[ S^T y = \text{vec} [\Delta R] \]

\[ J^T y = (S [V^T \otimes I_m \ I_n \otimes U])^T y \]

\[ = \begin{bmatrix} V \otimes I_m & I_n \otimes U^T \end{bmatrix} S^T y \]

\[ = \begin{bmatrix} V \otimes I_m \\ I_n \otimes U^T \end{bmatrix} \text{vec} [\Delta R] \]

\[ = \begin{bmatrix} (V \otimes I_m) \text{vec} [\Delta R] \\ (I_n \otimes U^T) \text{vec} [\Delta R] \end{bmatrix} \]

\[ = \begin{bmatrix} \text{vec} [\Delta R V^T] \\ \text{vec} [U^T \Delta R] \end{bmatrix} \]

Since \( \Delta R \) has only \( l \) non-zero elements, this operation also has complexity \( O(lk) \).

5.1.2 Line search

The simplest way to determine a step size that ensures that the value of the cost function will decrease is to perform a search on the line defined by the current point and the direction \( \begin{bmatrix} \Delta u \\ \Delta v \end{bmatrix} \). More precisely, one has to solve the following problem:

\[ \arg\min_\lambda \| M \odot ((U + \lambda \Delta U)(V + \lambda \Delta V) - \overline{X}) \|^2_\rho \]  

(46)

Usually line search uses a black-box style minimization method like golden-ratio search. In [5] Hartley and Schaffalitzky show that for any given line the cost function is a 4th degree polynomial. This can be used to perform the line search analytically. Line search can be reformulated as:

\[ \arg\min_\lambda P(\lambda) \]

where

\[ P(\lambda) = \| M \odot ((U + \lambda \Delta U)(V + \lambda \Delta V) - \overline{X}) \|^2_\rho \]

\[ = \| M \odot (UV - \overline{X} + \lambda(U \Delta V + \Delta UV) + \lambda^2 \Delta U \Delta V) \|^2_\rho \]

\[ = \| M \odot (A + \lambda B + \lambda^2 C) \|^2_\rho \]

\[ = \sum_{(i,j) | a_{ij} \neq 0} (a_{ij} + \lambda b_{ij} + \lambda^2 c_{ij})^2 \]  

(47)

and

\[ A = UV - \overline{X} \]

\[ B = U \Delta V + \Delta UV \]

\[ C = \Delta U \Delta V \]
\[ P(\lambda) = \sum_{(i,j)|m_{ij}=1} (a_{ij} + \lambda b_{ij} + \lambda^2 c_{ij})^2 \]

\[ = \sum_{(i,j)|m_{ij}=1} (a_{ij}^2 + \lambda^2 b_{ij}^2 + \lambda^4 c_{ij}^2 + 2\lambda a_{ij} b_{ij} + 2\lambda^2 a_{ij} c_{ij} + 2\lambda^3 b_{ij} c_{ij}) \]

\[ = \sum_{(i,j)|m_{ij}=1} (a_{ij}^2 + 2a_{ij} b_{ij} + (b_{ij}^2 + 2a_{ij} c_{ij})\lambda^2 + 2b_{ij} c_{ij} \lambda^3 + c_{ij}^2 \lambda^4) \]

\[ = \lambda^0 \sum_{(i,j)|m_{ij}=1} a_{ij}^2 \]

\[ + \lambda^1 \sum_{(i,j)|m_{ij}=1} 2a_{ij} b_{ij} \]

\[ + \lambda^2 \sum_{(i,j)|m_{ij}=1} (b_{ij}^2 + 2a_{ij} c_{ij}) \]

\[ + \lambda^3 \sum_{(i,j)|m_{ij}=1} 2b_{ij} c_{ij} \]

\[ + \lambda^4 \sum_{(i,j)|m_{ij}=1} 2c_{ij}^2 \]

So \( P(\lambda) \) is a polynomial of degree 4 that can be easily minimized. Calculating the coefficients of \( P(\lambda) \) can be done in \( O(\ell k) \) time. This is much faster than golden-ratio search and other black-box optimization routines, that require multiple evaluations of the cost function, each of which costs \( O(\ell k) \) steps.

### 5.1.3 Levenberg-Marquardt and Trust-Region-Reflective

We also created implementations based on the Levenberg-Marquardt and Trust-Region-Reflective algorithms available in the MATLAB optimization toolbox. Both of these algorithms extend Gauss-Newton by evaluating the quality of the linear approximation and dynamically adjusting the step size and direction in order to guarantee that the function value will decrease without performing line search. In terms of recovery rate all the approaches (GN with line search, LM and TRR) performed similarly. This is why we chose to present only the results for the Levenberg-Marquardt method, which is also the most popular non-linear least squares approach used for bundle adjustment in computer vision. A problem in MATLAB’s implementation of LM is that it is hard-coded to use a direct solver for the step direction. For the relatively small problem sizes in most of the experiments this did not cause a significant slowdown but for our tests on large-rank matrices we had to use TRR instead. All three implementations are available in the source code package that is part of this thesis.

### 5.2 Alternating Least Squares (ALS)

ALS is an application of the coordinate descent method for low-rank factorization problem. More specifically, ALS is a cyclic block-coordinate descent method. It alternatively optimizes the values of \( U \) and \( V \) while keeping the
other one fixed and can be implemented as alternating applications of equations (36) and (37). See Alg. 1.

Algorithm 1
Alternating Least Squares: Naive version
1: Initialize $\mathbf{u}$ and $\mathbf{v}$.
2: repeat
3:  $\mathbf{u} \leftarrow \hat{\mathbf{u}} = (F^T F)^{-1} F^T \mathbf{x}$
4:  $\mathbf{v} \leftarrow \hat{\mathbf{v}} = (G^T G)^{-1} G^T \mathbf{x}$
5: until converged

A more efficient version (Alg. 2) decomposes the update in steps 3 and 4 into several smaller linear least-squares problems. A key observation is that the elements in the $i$-th row of $\mathbf{X}$ depend only on the $i$-th row of $\mathbf{U}$ and the matrix $\mathbf{V}$. This means that one can solve for each row of $\mathbf{U}$ independently using a standard linear least-squares solver. Missing elements are handled by masking the corresponding elements of $\mathbf{x}_i$ and columns of $\mathbf{V}$. Analogously one can find the optimum for each column of $\mathbf{V}$ independently. When solving for the rows of $\mathbf{U}$ we have $m$ independent linear least squares problems of sizes $l_i \times k$, where $l_i$ is the number of observed elements in the $i$-th row. The $i$-th problem has complexity $O(l_i k^2)$. Since $l = \sum_i l_i$, the time required to compute $\mathbf{U}$ is $O(k^2)$. Analogously the time for computing the optimal value of $\mathbf{V}$ is also $O(k^2)$.

Algorithm 2
Alternating Least Squares: Optimized version
1: Initialize $\mathbf{U}$ and $\mathbf{V}$.
2: repeat
3:  for $i = 1$ to $m$ do
4:    mask $\leftarrow$ indexes of observed elements in the $i$-th row of $\mathbf{X}$
5:    $\mathbf{u}_i \leftarrow \text{lsqlin}(\mathbf{V}^T \cdot \text{mask}, \mathbf{X}_{i, \text{mask}})$
6:  end for
7:  for $j = 1$ to $n$ do
8:    mask $\leftarrow$ indexes of observed elements in the $j$-th column of $\mathbf{X}$
9:    $\mathbf{v}_j \leftarrow \text{lsqlin}(\mathbf{U}_{\cdot, \text{mask}}, \mathbf{X}_{\text{mask}, j})$
10: end for
11: until converged

5.3 Wiberg’s algorithm

Wiberg’s algorithm is an instance of the variable projection method specialized for low-rank matrix approximation. The variable projection method is a method for solving least-squares problems in which the model is linear in part of the parameters. More details about it can be found in [3].

In the case of low-rank matrix approximation the model is linear in both $\mathbf{u}$ and $\mathbf{v}$ separately. When applying a variable projection method one can decide which part of the parameters are to be dropped. Due to the importance of the Wiberg algorithm in our experiments, we provide a detailed derivation of this algorithm. The derivation follows closely the presentation in [7]. Although we use a similar notation, we parametrize the cost function using $\mathbf{u}$ while Okatani
and Deguchi use $v$. This is entirely a notation difference and the two derivations are equivalent.

**Derivation of the Inner LS-Problem**

The main point in this algorithm is the re-parametrization of the cost function. From (37) you know that for all local minima of $e(u, v)$ it holds that $v = \hat{v}(u)$, where $\hat{v}(u)$ is the least squares solution for $v$ for the given $u$. This allows us to restrict the optimization procedure to work on the subspace defined by this equation. This leads to two equivalent ways of writing the optimization problem that we are solving. The first one is:

$$
\text{minimize } e(u, v) \\
\text{s.t. } v = \hat{v}(u) = (G^T G)^{-1} G^T x
$$

An alternative view is that we are minimizing a modified cost function:

$$
e(u) = e(u, \hat{v}(u)) = \rho^T \rho = \|\rho\|_2^2
$$

where $\rho$ is the residual as a function of $u$:

$$
\rho = G\hat{v} - \bar{x} \\
= -(I - G(G^T G)^{-1} G^T)x \\
= -Q_G\bar{x}
$$

$Q_G$ is a projection to the subspace orthogonal to the columns of $G$.

The Wiberg algorithm uses Gauss-Newton steps to optimize the cost function $\rho$ and needs to compute the Jacobian $\frac{\partial \rho}{\partial u}$, which is not the same as $\frac{\partial r}{\partial u}$. The connection between the them is given by:

$$
J = \frac{\partial \rho}{\partial u} = \frac{\partial r}{\partial u} + \frac{\partial r}{\partial v} \frac{\partial \hat{v}}{\partial u}
$$

We know that $\frac{\partial r}{\partial v} = F$ and $\frac{\partial r}{\partial u} = G$. The problem is computing $\frac{\partial r}{\partial v}$. In order to do that we will look at the the partial derivative of the original cost function $e(u, v)$ with respect to $v$. Let $f(u, v) = \frac{\partial e}{\partial v}$:

$$
f(u, v) = \frac{\partial e}{\partial v} \\
= \frac{\partial (r^T r)}{\partial v} \\
= 2 \frac{\partial r^T r}{\partial v} \\
= 2G^T r
$$
The construction of \( \hat{v}(u) \) ensures that:

\[
\phi(u) = f(u, \hat{v}(u)) = \bar{u} \quad \forall u 
\]

\[
\phi(u) = 2G^T \rho(u) 
\]

So \( \phi \) which is a function of \( u \) is the constant zero. This implies that its Jacobian with respect to \( u \) must be zero:

\[
\bar{u} = \frac{\partial \phi}{\partial u} = \frac{\partial G^T \rho(u, \hat{v})}{\partial u} = \frac{\partial G^T}{\partial u} \rho + G^T \frac{\partial \rho}{\partial u} 
\]

(56)

In the spirit of the Gauss-Newton algorithm, one can assume that the norm of the residual \( \rho \) is small and drop the term \( \frac{\partial G}{\partial u} \rho \).

\[
\bar{u} \approx G^T \frac{\partial \rho}{\partial u} 
\]

(57)

It follows that:

\[
\frac{d\hat{v}}{du} \approx -(G^TG)^{-1}G^TF 
\]

(58)

Which can be substituted in (52) in order to get an approximate formula for the Jacobian:

\[
J = \frac{\partial r}{\partial u} + \frac{\partial r}{\partial \hat{v}} \frac{\partial \hat{v}}{\partial u} 
\]

\[
= F + G \frac{\partial \hat{v}}{\partial u} 
\]

(59)

\[
\approx F + G(-G^TG)^{-1}G^TF = (I - G(G^TG)^{-1}G^T)F 
\]

\[
= Q_GF 
\]

Having calculated the residual in (51) and the approximate Jacobian in (59) one can perform a Gauss-Newton step. The increment is computed by solving:

\[
\|J\Delta u + r\|_2 \rightarrow \min 
\]

(60)

Which is equivalent to:

\[
\|Q_GF\Delta u - Q_Gx\|_2 \rightarrow \min 
\]

(61)

In [7] Okatani and Deguchi prove that the matrix \( Q_GF \) is rank deficient with rank at most \((m-k)k\). This means that there exists a linear subspace of equivalent least squares solutions. Since the solution represents a step and the system is a local approximation of the original problem, it is reasonable to limit the norm of the step in some way. This leads to various ways to regularize the problem.
Solving the Inner LS-Problem

The size of this least squares problem is \( l \times mk \). A typical large structure-from-motion problem can have thousands of rows \( m \) and millions of observations \( l \). This is why solving for the step direction is usually the most compute-intensive part of the algorithm and requires special attention.

Okatani and Deguchi recommend using the Moore-Penrose pseudo-inverse to solve this problem. In an under-determined system, multiplying with the pseudo-inverse leads to the solution that has minimal \( l^2 \) norm. This can be computed using the truncated SVD of the Jacobian. The problem with this approach is that computing the SVD requires \( O(l^2mk + lm^2k^2) \) steps which makes it unfeasible for large problems. An implementation that uses QR decomposition has a complexity \( O(lm^2k^2) \) which is a lot faster but still has a quadratic dependence on \( m \).

Instead we used LSQR, which is an iterative method for solving linear least-squares problems. It does not require explicit representation of the matrix but only procedures that multiply by the Jacobian and its transpose. Since \( F \) and \( G \) are sparse these multiplications can be computed efficiently as several successive sparse matrix-vector multiplications. One can decompose the multiplication \( J\Delta u \) into two successive matrix-vector multiplications: \( Q_G(F\Delta u) \). The first one \( F\Delta u \) takes \( O(lk) \) steps due to the sparsity of \( F \). Let \( z = F\Delta u \). \( Q_G \) is a projection to the subspace orthogonal to the column space of \( G \). From this:

\[
Q_Gz = (I_l - P_G)z = z - P_Gz
\]

where \( P_G \) is the projector to the column space of \( G \). But \( G \) has a block diagonal structure and can be orthogonalized in \( O(lk^2) \) steps because each block can be orthogonalized separately from the others. An orthogonal basis can be easily assembled from the individual orthogonal bases of the blocks. Let \( O_G \) be an orthonormal basis of the column space of \( G \). Then:

\[
Q_Gz = z - P_Gz = z - O_GO_G^Tz
\]

and this can be computed in \( O(lk) \) steps. \( O_G \) does not change between iterations of the LSQR algorithm and can be precomputed. This way the complexity of one LSQR iterations is \( O(lk) \).

In [4] Hanke shows that iterative methods like LSQR and CGLS can be used to solve ill-conditioned least squares problems and they give solutions similar to the ones obtained by truncated SVD. The key point is that these methods have the so-called semi-convergence property. They initially converge to the solution and later diverge driven by the noise and numerical errors. This means it is important to terminate the iterations early enough. Hanke observed that the number of iterations is in practice smaller than the optimal number of components of the truncated SVD. That is why we chose to run LSQR for a maximum of \((m - k)k\) iterations. We do not have a hard theoretical justification for this number but we have observed in experiments that it works at least as well as the truncated SVD approach. Each Wiberg outer iteration is limited to \( O(m - k)k \) internal LSQR iterations and each of them has complexity \( O(lk) \). Computing \( O_G \) takes additional \( O(lk^2) \) operations per outer iteration. Thus the complexity of one Wiberg outer iteration in our implementation is \( O(lmk^2) \).
In practice the LSQR algorithm reaches the needed precision and the inner iterations stop early so this bound is actually pessimistic.

**Line Search**

As described in [7], Wiberg’s algorithm does not include any form of step size control. This allows the algorithm to do steps that increase the value of the cost function. Unfortunately the scheme for doing the line search analytically, presented in Sec. 5.1.2, can not be used for this model because $\hat{\mathbf{v}}(\mathbf{u})$ is a non-linear function. We added a line search procedure using MATLAB’s built-in function ”fminunc”, which generally reduces the number number of iterations at the cost of making each iteration more expensive. The cost of a line search can be significant since the complexity of evaluating the cost function at a single point is $O(k^2)$. In practice adding line search can double the time per iteration, which may not justify the savings in the number of iterations. We used it because it is the simplest way to ensure that the algorithm will converge, although a trust-region based approach may be faster.

**Algorithm 3** Wiberg’s algorithm

1. Initialize $\mathbf{u}$.
2. repeat
3. $\mathbf{G} \leftarrow \mathbf{S}(\mathbf{I}_n \otimes \mathbf{U})$
4. $\mathbf{v} \leftarrow \hat{\mathbf{v}}(\mathbf{u}, \mathbf{x})$ (computed in the same way as in Alg. 2)
5. $\mathbf{F} \leftarrow \mathbf{S}(\mathbf{V}^T \otimes \mathbf{I}_n)$
6. $\mathbf{O}_G \leftarrow \text{orth}(\mathbf{G})$
7. $\mathbf{rhs} \leftarrow \mathbf{x} - \mathbf{O}_G(\mathbf{O}_G^T \mathbf{x})$
8. $\Delta \mathbf{u} \leftarrow \text{LSQR}(J\text{multfunc}, J^T\text{multfunc}, \mathbf{rhs})$
9. $\alpha = \text{argmin}_\alpha \varepsilon(\mathbf{u} + \alpha \Delta \mathbf{u})$
10. $\mathbf{u} \leftarrow \mathbf{u} + \alpha \Delta \mathbf{u}$
11. until converged

**Algorithm 4** Jmultfunc

1. $\mathbf{z} \leftarrow \mathbf{F}\mathbf{x}$
2. return $\mathbf{z} - \mathbf{O}_G(\mathbf{O}_G^T \mathbf{z})$

**Algorithm 5** JTmultfunc

1. return $\mathbf{F}^T(\mathbf{x} - \mathbf{O}_G(\mathbf{O}_G^T \mathbf{x}))$

**5.4 Other Algorithms**

In the process of preparing this thesis we looked into several other algorithms. The topic of low-rank matrix factorization is especially popular in the collaborative filtering community. We tried two methods that have been successfully used to train a factorized model of the Netflix dataset. Both methods performed worse than the approaches that were described and this is why we left them out.
of the comparison. We include a short description of our observations for the sake of completeness. Implementations are included in the source code package that is part of this thesis.

**Gradient Descent and Stochastic Gradient Descent (SGD)**

Gradient descent performs a line search in the direction of steepest descent ($J^T r$ for least squares problems). This can be computed very efficiently - $O(lk)$.

SGD uses a single data point error function. In each iteration it samples from the empirical distribution of the data and makes a small step in the opposite direction of the gradient. The size of the step is controlled by a "learning rate" parameter. Because a single data point depends only on one row of $U$ and one column of $V$ the cost of such a step is $O(k)$. This also makes the algorithm really fast and simple to program.

Both Gradient descent and SGD had similar but worse recovery rates and convergence speed than Alternating Least Squares. This why we decided to leave them out of the comparison. However, both algorithms have the advantage of being linear in the rank parameter instead of quadratic. ($O(lk)$ instead of $O(lk^2)$). This may be an advantage when constructing factorizations of a larger rank. For structure from motion problems the rank is usually 3 or 4 and this is not really an advantage.

**Bayesian Probabilistic Matrix Factorization**

Bayesian Probabilistic Matrix Factorization by Salakhutdinov and Mnih [9] is a probabilistic model of matrix factorization that can be sampled using Markov Chain Monte Carlo techniques. In the original paper it was sampled using a Gibbs sampler which loosely resembles cyclic block coordinate descent with some randomness added in each step. This makes the method very similar to Alternating Least Squares. This was also observed in experiments. Additionally in structure from motion tasks one is usually targeting a point estimate for the camera projection matrices which is incompatible with the set of samples (separate factorizations) that is provided by MCMC.
6 Synthetic Experiments

The performance of the algorithms was evaluated on synthetic data. Each method was applied to a set of random low-rank matrices. In each run a subset of the elements of the random matrix was given as an input to a recovery algorithm. The output was interpreted as an attempted recovery of the original matrix. Such setup allowed us to compare the results of the algorithms with the ground truth matrix. The main goals of this evaluation were to answer the following questions:

• How often do different algorithms converge to a the global minimum?
• How much iterations do they need to converge?
• How many observations are necessary in order to recover the original matrix reliably?

The random synthetic problems were generated using the following procedure:

**Algorithm 6 Create random synthetic experiment.**

1: Generate $U$ and $V$ using the full dimensionality $(m + n)k$. Elements are i.i.d. samples from $\mathcal{N}(0, 1)$.
2: Use QR decomposition to orthogonalize and normalize $U$ and $V^T$.
3: Set $X = USV$. Where $S$ is a fixed diagonal matrix containing the singular values.
4: Add normally distributed noise. $X_{\text{noise}} = X + \text{randn}(m, n)\sigma_{\text{noise}}$.
5: Sample the elements of $X_{\text{noise}}$ to get the observed matrix $X$.

Using normal distribution in step 1 ensures that the column space of $U$ and the row space of $V$ are uniformly distributed. In step 2 QR decomposition is used to construct orthonormal bases of those subspaces. Then fixed singular values are introduced and the matrix $X$ is constructed. By controlling the singular values one can control the magnitude of the matrix elements. It also allows to construct degenerate cases in which the singular values are unbalanced. In all experiments we enforced $\|UV\|_F^2 = mn$ by setting the norm of the singular values vector to $\sqrt{mn}$. This way the root mean square of the matrix elements is 1. In the base experiment the magnitude of the noise is $\sigma_{\text{noise}} = 0.01$ so that the expected value of $\text{RMSE}(X, X_{\text{noise}})$ is 0.01. This means that the global minimum of the cost function is less than or equal to 0.01 and close to it. This allows to roughly classify if a particular run has converged to the global minimum ($\leq 0.01$) or to a local minimum ($\geq 0.01$).

Two different algorithms were used in the last step to sample matrix elements:

• Uniform sampling.
  This approach selects, uniformly at random, from all possible subsets of the matrix elements with the given size. If a row or a column has less samples than the rank of the matrix, additional elements are included by uniformly sampling the not-included elements in the row/column.
• Diagonal sampling.
  This approach samples uniformly in a band around the main diagonal of the matrix. This method is designed to simulate an observation pattern that is common in real-world structure from motion data. Our results showed that such observation structure can have significant effects on the chances of some algorithms to recover the original matrix. The method first removes all elements outside the diagonal band and then calls the uniform sampler on the rest of the elements. The band width is selected to be the minimal possible that has no less than 1.5 times the requested number of elements.

The number of observed elements is selected uniformly at random in the range 1.1 to 6 observations per degree of freedom. For each fixed values of the parameters(matrix size, rank, singular valus, noise magnitude) 200 test cases were generated. Then each algorithm was ran on each test case.

6.1 Convergence criteria.

All algorithms were run for a maximum of 100 iterations or until they converged. A run was considered converged if:

- the root mean squared error(RMSE) measured on the training data is less than $10^{-2}$ OR

- the change in the RMSE has been lower than $10^{-5}$ for 5 consecutive iterations.

In general, comparison with limited number of iterations can be misleading since some algorithms need more iterations to converge than others. Also some(i.e. ALS) have much lower iterations times than others.That is why we ran a set of experiments in which the maximum number of iterations was increased to 1000. This did not lead to a significant change of the results. After 100 iterations, the algorithms converged reasonably close to a minima in most cases and an increased number of iterations did not change the results considerably. This is described in detail in Sec. 6.3.1.

6.2 Plotting the results

The results are presented as scatter-plots in which each point represents the result of a single run of a recovery algorithm on a randomly generated problem. The problems and their corresponding initial values are the same for all compared algorithms. A blue circle marks a converged run. Red cross means that the algorithm was terminated because the maximum number of iterations had been reached.

On the horizontal axis is plotted the number of observed elements of the matrix $\mathbf{X}$ divided by the number of degrees of freedom $(m + n - k)$. The vertical axis represents the base-10 logarithm of the RMSE. The are two types of error in the plots.The first one is the training error $\frac{1}{r} \| M \odot (\mathbf{X} - \mathbf{UV}) \|^2_F$ and the second
Table 2: Synthetic experiment parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>matrix size</td>
<td>$100 \times 300$</td>
</tr>
<tr>
<td>rank</td>
<td>$3$</td>
</tr>
<tr>
<td>$\sigma_{\text{noise}}$</td>
<td>$10^{-2}$</td>
</tr>
<tr>
<td>maximal number of iterations</td>
<td>$100$</td>
</tr>
<tr>
<td>convergence criteria</td>
<td>$\text{RMSE} &lt; 10^{-2}$ or $(</td>
</tr>
</tbody>
</table>

one is the error with respect to the ground truth matrix $(\frac{1}{m} \| X - UV \|_F^2)$. We set $\sigma_{\text{noise}} = 10^{-2}$ which causes the expected value for the RMSE between $X$ and $\bar{X}$ to be $10^{-2}$. If the algorithm converged to a solution with a training error value less than $10^{-2}$ we consider it as converged to the global minimum. In all cases where the error to ground truth is less than $10^{-2}$ we consider the original matrix as recovered.

6.3 Results: Base experiment

For this experiment we used balanced singular values 110.6567, 99.5910 and 88.5253 (chosen to have the ratio 10:9:8). These values are chosen in a way that the data is spread in all three dimensions and the magnitude of the matrix 100 times larger than the expected magnitude of the noise. The plots of the training error are presented on page 29. The first row contains the results for uniform sampling and the second for diagonal sampling. The results show that the sampling structure has a dramatic effect on recovery abilities of ALS and LM. The number of cases that were recovered by these two algorithms is significantly lower in the diagonally sampled case, while there is no or very small difference for Wiberg’s algorithm. Also the results show that Wiberg’s algorithm greatly outperforms ALS and Levenberg-Marquardt by being able to recover almost all cases where the number of observed elements is larger than some value (around 2.5 for this experiment). This coincides with the observations of Okatani and Deguchi in [7].

6.3.1 Number of iterations

On page 30 the reader can see similar plots that show the number of iterations that the algorithms performed before converging. Results show that the number of iterations necessary is generally larger for diagonally sampled cases. Alternating least squares and Wiberg’s algorithm require several times more iterations in the diagonally sampled cases, Levenberg-Marquardt also shows a small increase. In both sampling schemes the number for iterations decreased when the number of observation increased.

Because of the large part of runs that did not converge we repeated the experiment increasing the maximum number of iterations to 1000. The results are presented on pages 32 and 33. With the increased limit all the runs converged but this did not increase the number of runs that converged to the global...
minimum. The behavior commonly observed during experiments was that depending on the starting point the algorithm either converged very fast to the global minimum or flat-lined for a long time before getting stuck at a local minimum. This caused the two distinct point clouds in the error and iteration plots. This experiment shows that in most cases these flat-lined cases are hopeless and it is reasonable to terminate the algorithm early if it does not converge in some number of iterations (i.e. 100). In such situations it is more likely to reach the global optimum by restarting the optimization from another random point.

6.3.2 Generalization error

On page 31 is presented the error to ground truth. It shows that solutions which correspond to a global minimum of the cost function generalize well and generalization error decreases as the number of observations increases. In some cases with low number of observations Wiberg’s algorithm converged to the global minimum of the cost function but the solution does not generalize well to unseen data. This is visible in the plots as runs where the final value of the training error at the global minimum is lower than $10^{-2}$ but the error to ground truth matrix is larger than $10^{-2}$. This is a problem of the used least squares model and nothing to do with the optimization procedure. In Sec. 7.5 we propose a way to regularize structure-from-motion problems. Initially we expected that this can counter overfitting but we could not observe this in our experiments.
Figure 1: Training error
Figure 2: Number of iterations
Figure 3: Error to ground truth
Figure 4: 1000 iterations limit. Training error.
Figure 5: 1000 iterations limit. Number of iterations performed.
6.4 Unbalanced singular values

In real-world structure from motion problems, there are sometimes large differences in the magnitude of the singular values. For example in the dinosaur dataset without normalization the largest singular value is 5 to 10 times larger than the rest. In order to study the effect of such dis-balance on the recovery process we performed an experiment in which the largest non-zero singular values have the ratio 100:10:1. In such situation the component that corresponds to the smallest singular value can be mistaken for noise. That is why no noise was added. Because of that the global minimum has zero value of the cost function and the threshold for stopping the optimization was reduced to $10^{-6}$.

The results are presented on page 35. They show that such dis-balance in the singular values can reduce the recovery performance of all algorithms, especially in the diagonally sampled case. For all algorithms there were more runs that converged to local minima. ALS never converged on diagonally sampled cases. These results show that when applying matrix factorization to real-world data care must be taken to normalize the input data in a way that will reduce the difference between singular values.

6.5 Large rank matrices

It was also interesting to see how the performance of the algorithms changes when the rank of the matrices increases. We performed an experiment in which the ground truth matrices are of rank 10 with singular values ratio 10:9:...:2:1. Because of the increased number of variables involved in the optimization, the Levenberg-Marquardt implementation in MATLAB ran out of memory and could not complete the experiment. Instead we included the Trust-Region-Reflective algorithm that is available in MATLAB, which uses an iterative method to solve for the step direction and needs less memory. The results are presented on page 36 (training error) and 37 (generalization error). It is important to note that in these plots the horizontal axis has a different scale due to the increased number degrees of freedom of the model. In this experiment Wiberg’s algorithm again outperformed ALS and LM. In all but a few runs it converged to the global minimum. However in cases when the number of observations is low such global minimum does not generalize well. ALS and LM also always reach the global minimum for runs with more than roughly 4 observation per degree of freedom for the diagonally sampled case but this corresponds to sampling in a diagonal band that covers more than 75% of the matrix. In this case the diagonal structure is almost lost.
Figure 6: Unbalanced singular values. Training error.
Figure 7: Rank 10 (training error)
Figure 8: Rank 10 (generalization error)
7 Application to Affine Structure from Motion problems

Structure from Motion is the problem of recovering the three-dimensional structure of a scene from photographs taken from multiple viewpoints. Usually local features are extracted from each photograph and later matched between the different views. The result is a set of trajectories of local features, which are usually incomplete due to occlusions and matching errors. Using those trajectories one can recover the world coordinates of the features and calibration of the cameras, which can be used for further processing of the scene. The problem of recovering the camera parameters and sparse scene structure is closely related to low-rank matrix factorization.

7.1 Affine factorization model

The affine camera is a camera model that assumes that all observed points are approximately at the same distance from the camera. The projection equation is:

\[
\begin{bmatrix}
  x \\
  y \\
\end{bmatrix} = P_{2 \times 4} \begin{bmatrix}
  X \\
  Y \\
  Z \\
  1
\end{bmatrix}
\]

(64)

Where \( P \) is the \( 2 \times 4 \) camera projection matrix. \( \begin{bmatrix} X & Y & Z \end{bmatrix}^T \) are the world coordinates of a point and \( \begin{bmatrix} x & y \end{bmatrix}^T \) are the corresponding camera coordinates of its projection. This equation can be generalized to multiple cameras and multiple scene points by stacking the camera matrices vertically and concatenating the point coordinates horizontally.

\[
\begin{bmatrix}
  x_{11} & x_{12} & \cdots & x_{1n} \\
  y_{11} & y_{12} & \cdots & y_{1n} \\
  x_{21} & x_{22} & \cdots & x_{2n} \\
  y_{21} & y_{22} & \cdots & y_{2n} \\
  \vdots & \vdots & \ddots & \vdots \\
  x_{f1} & x_{f2} & \cdots & x_{fn} \\
  y_{f1} & y_{f2} & \cdots & y_{fn}
\end{bmatrix}_{2f \times n} = \begin{bmatrix}
  P_1 \\
  P_2 \\
  \vdots \\
  P_f
\end{bmatrix} \begin{bmatrix}
  X_1 & X_2 & \cdots & X_n \\
  Y_1 & Y_2 & \cdots & Y_n \\
  Z_1 & Z_2 & \cdots & Z_n \\
  1 & 1 & \cdots & 1
\end{bmatrix}_{f \times 4}{4 \times n}
\]

(65)

This model can be interpreted as a factorization of rank 3 with per-row offsets or as a degree 4 factorization with the last row of \( \mathbf{V} \) fixed to all ones. In a structure-from-motion problem the observed trajectories can be arranged in a matrix and one can compute the factorization on the right hand side. Such reconstruction is ambiguous because there is no fixed coordinate system in which to describe the world coordinates, but can be easily upgraded to one in which the coordinate system is metric.

For simplicity we will note the matrix containing the projections with \( \mathbf{X} \), the stacked projection matrices as \( \mathbf{\tilde{U}} \) and the concatenated coordinates with \( \mathbf{\tilde{V}} \).
similar to the low-rank factorization model. They can be decomposed as:

\[
\tilde{U} = \begin{bmatrix} U & z \end{bmatrix} \quad \tilde{V} = \begin{bmatrix} V^T \end{bmatrix}
\]  
\( (66) \)

\( U \) and \( V \) are components of a 3-dimensional low-rank factorization and \( z \) are the per-row offsets. The model can be written as:

\[
X = \tilde{U}\tilde{V} = \begin{bmatrix} U & z \end{bmatrix} \begin{bmatrix} V^T \end{bmatrix} = UV + zT^T
\]  
\( (67) \)

This model can be vectorized similar to the low-rank factorization model:

\[
x = S vec[X] = \tilde{F}\tilde{u} = Gv + w
\]

where

\[
\tilde{F} = S(\tilde{V}^T \otimes I_m)
\]

\[
G = S(I_n \otimes U)
\]

\[
\tilde{u} = vec[\tilde{U}]
\]

\[
v = vec[V]
\]

\[
w = S vec[z_1^T]
\]

\( \tilde{F} \) and \( G \) are the Jacobians of the model with respect to \( \tilde{u} \) and \( v \). \( w \) is a vector of per-data point offsets that are derived from \( z \). Similarly to the general low-rank factorization, this model is bilinear. This means that one can formulate the least squares cost function and fit the model to data using the same algorithms. ALS, LM and Wiberg’s algorithm can be adapted operate on affine structure-from-motion data. The complexity of the algorithms does not change. Also the experiments on synthetic data presented in Sec. 8 show that they also have similar properties to their non-affine variants.

### 7.2 Alternating least squares.

ALS is trivial to modify to include the per-camera offsets. The procedure becomes alternation between optimizing \( \tilde{U} \), which consists of \( U \) and the offsets \( z \), and optimizing \( V \). The offsets have to be subtracted when optimizing \( V \).
Algorithm 7 Alternating Least Squares: affine camera version

1: Initialize $\tilde{U}$ and $V$.
2: \textbf{repeat}
3: \hspace{1em} $\tilde{V} \leftarrow \begin{bmatrix} V \end{bmatrix}^T$
4: \hspace{1em} \textbf{for} $i = 1$ to $m$ \textbf{do}
5: \hspace{2em} mask \leftarrow indexes of observed elements in the $i$-th row of $X$
6: \hspace{2em} $u_i \leftarrow \text{lsqlin}(\tilde{V}_i \text{mask}, \begin{bmatrix} X_i \text{mask} \end{bmatrix})$
7: \hspace{1em} \textbf{end for}
8: $U \leftarrow \tilde{U}_{1:3,4}$
9: $z \leftarrow \tilde{U}_{4}$
10: \textbf{for} $j = 1$ to $n$ \textbf{do}
11: \hspace{1em} mask \leftarrow indexes of observed elements in the $j$-th column of $X$
12: \hspace{2em} $v_j \leftarrow \text{lsqlin}(U_{mask,:}, \begin{bmatrix} X_{mask,j} - z_{mask} \end{bmatrix})$
13: \hspace{1em} \textbf{end for}
14: \textbf{until} converged

7.3 Gauss-Newton based methods.

In order to adapt GN-based algorithms one has to compute the Jacobian of the residual with respect to the parameters. If the problem is parametrized with the vector $\begin{bmatrix} \tilde{u} & v \end{bmatrix}$ the Jacobian is:

$$
\frac{\partial r}{\partial \begin{bmatrix} \tilde{u} \n v \end{bmatrix}} = \begin{bmatrix} \frac{\partial r}{\partial \tilde{u}} & \frac{\partial r}{\partial v} \end{bmatrix} = \begin{bmatrix} \frac{\partial (\tilde{F}\tilde{u} - x)}{\partial \tilde{u}} & \frac{\partial (Gv + S\text{vec}[z^T] - x)}{\partial v} \end{bmatrix} = \begin{bmatrix} \tilde{F} & G \end{bmatrix}
$$

7.4 Wiberg’s algorithm.

Wiberg’s algorithms is derived in the same way as the version without offsets. The formulas for the residual and the approximate Jacobian are:

$$
r = Q_G(x - w) \quad J \approx Q_G\tilde{F}
$$

Where $G$ is again block-diagonal and can be easily orthogonalized. The algorithm was implemented using the LSQR algorithm to compute the step direction in the same way as the non-affine variant.

7.5 Orthogonality constraints

An affine camera matrix can be decomposed in the following way:

$$
P = \begin{bmatrix} u_1^T & z_1 \\ u_2^T & z_2 \end{bmatrix}
$$

where $u_1^T$ and $u_2^T$ are vectors that define the axes of the camera projection plane and $(z_1, z_2)$ is an offset in this plane. Real-world cameras have axes that are approximately orthogonal and uniformly scaled. The factorization representation
is over-parametrized because it allows cameras that have non-uniformly scaled and non-orthogonal axes. To counter this, soft constraints were added to the cost function in the form of additional residuals which should be minimized in least squares sense. This augmented cost function minimization is again a least-squares problem and can be minimized using Gauss-Newton style methods. As the reader will be able to see in next section, this caused Wiberg’s algorithm to reach the global minimum more often and in less iterations.

The residual with orthogonality constraints is defined as:

$$r = \begin{bmatrix} \sqrt{\frac{\lambda}{m}} r_m \\ \sqrt{\frac{1}{m}} r_c \end{bmatrix}$$

$$r_m = Q_G (\mathbf{x} - \mathbf{w}) \quad \text{(model residual)}$$

$$r_c = \begin{bmatrix} u_1^T u_2 - 0 \\ u_1^T u_1 - 1 \\ u_2^T u_2 - 1 \\ u_3^T u_4 - 0 \\ u_3^T u_3 - 1 \\ u_4^T u_4 - 1 \\ \vdots \\ u_{m-1}^T u_m - 0 \\ u_{m-1}^T u_{m-1} - 1 \\ u_m^T u_m - 1 \end{bmatrix} \quad \text{(orthogonality constraints residual)}$$

There are two sets of residuals that are involved. First $r_m$ that corresponds to how well the model matches the data. It is weighted by $\frac{1}{\sqrt{\lambda}}$ in order to make the value independent of the number of residuals. The second set $r_c$ corresponds to the orthogonality and scale constraints for the camera projection matrices. There are 3 constraints per camera. If they are matched exactly then then the camera has orthogonal and uniformly scaled axes. The constraint residuals are scaled by $\sqrt{\frac{\lambda}{m}}$. This ensures that the value of the error function will be independent of the number of cameras. $\lambda$ is a hyper-parameter that controls the trade-off between fitting of the model and fitting the orthogonality constraints.

The corresponding least-squares cost function is:

$$E(\tilde{U}, V) = \frac{1}{t} \| r_m \|_2^2 + \frac{\lambda}{m} \| r_c \|_2^2$$

$$= \frac{1}{t} \left\| \mathbf{M} \odot \begin{bmatrix} U^T \\ T^T \end{bmatrix} - \mathbf{X} \right\|_F$$

$$+ \frac{\lambda}{m} \sum_{i=1}^{m} \left( (u_{2i-1}^T u_{2i})^2 + (u_{2i-1}^T u_{2i} - 1)^2 + (u_{2i}^T u_{2i} - 1)^2 \right)$$

(73)

This cost function can be re-parametrized as a function of $\tilde{U}$ only in the spirit of Wiberg’s algorithm. When $\tilde{U}$ is fixed the value of the $r_c$ is also fixed and the optimal value for $V$ is the same as in the non-constrained case. The Jacobian
can be split in two parts that correspond to the model and to the constraints.

\[ J = \begin{bmatrix} J_m \\ J_c \end{bmatrix} = \begin{bmatrix} \sqrt{\frac{1}{m}} \partial \rho_m \\ \sqrt{\frac{1}{m}} \partial \rho_c \end{bmatrix} \]  

(74)

Where \( \rho_m \) and \( \rho_c \) are the two parts of the residual as function of \( \tilde{u} \) only similar to the derivation of Wiberg’s algorithm. The first part is the same of the non-regularizes version multiplied by \( \frac{1}{\sqrt{l}} \):

\[ \frac{\partial \rho_m}{\partial \tilde{u}} \approx \mathbf{Q}_G \hat{F} \]  

(75)

In the second part each row of \( \mathbf{U} \) influences only the three residuals that correspond to the same camera. The Jacobian is constructed using the following equations for each camera, setting all other elements to zero and multiplying by \( \sqrt{\frac{\lambda}{m}} \).

\[ \begin{align*}
\frac{\partial \rho_c^1}{\partial u_{2i-1}} & = u_{2i} \\
\frac{\partial \rho_c^1}{\partial u_{2i}} & = u_{2i-1} \\
\frac{\partial \rho_c^2}{\partial u_{2i-1}} & = 2u_{2i-1} \\
\frac{\partial \rho_c^3}{\partial u_{2i}} & = 2u_{2i}
\end{align*} \]  

(76)

Knowing the Jacobian, Gauss-Newton steps are performed in a way similar to the standard Wiberg algorithm. Step direction is solved using the LSQR method. Multiplication by the Jacobian is performed by splitting it into multiplication by the two parts of the Jacobian:

\[ Jy = \begin{bmatrix} J_m \\ J_c \end{bmatrix} y = \begin{bmatrix} J_m y \\ J_c y \end{bmatrix} = \begin{bmatrix} \sqrt{\frac{1}{m}} \partial \rho_m y \\ \sqrt{\frac{1}{m}} \partial \rho_c y \end{bmatrix} = \begin{bmatrix} \sqrt{\frac{1}{m}} \mathbf{Q}_G \hat{F} y \\ \sqrt{\frac{1}{m}} \partial \rho_c y \end{bmatrix} \]  

(77)

Multiplication by \( \mathbf{Q}_G \) is done using an orthogonalized version of \( \mathbf{G} \). \( \frac{\partial \rho_c}{\partial \tilde{u}} \) has \( O(m) \) non zero elements and does not increase the complexity of the algorithm. Furthermore the experiments show that adding the orthogonality constraints can cause Wiberg’s algorithm to converge in less iterations.

### 7.6 Normalization

The experiments on synthetic data showed that unbalanced singular values can decrease the chances of all algorithms to converge to the global minimum. To counter this we normalized the data. Points in each camera are first centered to their mean. After that the points observed by all cameras together are scaled in a way such that their standard deviation becomes one. The scaling is isotropic in in the x- and y-direction and the same scaling factor is used for all cameras.
This way minimizing the least-squares error on the normalized data is equivalent to minimizing it on the original dataset. For example on the dinosaur dataset, that we used for real-world data tests, this changes the ratio of the singular values of the recovered model from approximately 100:19:15:7 to approximately 100:63:32:28.
8 Experiments on Structure-from-Motion Data

Two experiments were performed on structure-from-motion (SFM) data. The first one consists of random samplings of a synthetic data in a way similar to the experiments described in Sec. 6. The other one is an evaluation of the performance of the algorithms on a real-world dataset.

Both experiments use the popular dinosaur dataset from http://www.robots.ox.ac.uk/~vgg/data/data-mview.html. The scene consists of a dinosaur figurine placed on a turntable and observed by single camera while rotating. Due to the rotational motion, the dataset has a diagonal observation structure (Fig. 11). It includes 319 features that were partially observed over 36 views. The provided camera calibration matrices were used to triangulate the position of each observed feature and trajectories that had mean triangulation error larger than 1 pixel were removed from the dataset in order to reduce the number of outliers. The filtered dataset contains 312 feature trajectories.

![A sample frame.](image)

**Figure 9: A sample frame.**

![Observation mask.](image)

**Figure 10: Observation mask.**

8.1 Evaluation on Synthetic SFM Data

We generated synthetic SFM data by re-projecting the triangulated feature points. The affine camera model was enforced by constructing affine approximations of the provided camera matrices. We added gaussian noise with standard deviation 1 to each element. Then the resulting full observation matrix was sampled using the uniform and diagonal sampling methods presented in Sec. 6. Multiple samplings of the same matrix were created and each algorithm was run on each of them.

The results are presented in Fig. 11. They are very similar to the results from the synthetic experiments in Sec. 6. ALS and LM need about the same amount of observations in order to recover the matrix and have significantly lower recovery rate in the diagonally sampled case. On the other hand Wiberg’s algorithm is able to do the same with less observations and has similar performance in the uniformly and diagonally sampled cases.
Figure 11: Results for synthetic affine structure-from-motion data. Training error.
8.2 Evaluation on Real-World SFM Data

ALS, LM and Wiberg’s algorithm were applied to the dinosaur dataset (the original measurements with the outliers removed). Because all algorithms guarantee only convergence to a local minima, each of them was run 200 times from random starting points. Additionally LM and Wiberg’s algorithm were tested with added orthogonality constraints, for different values of the regularization parameter $\lambda$.

The iteration limit was 300 for LM and Wiberg and 3000 for ALS which has much faster iterations. A run was considered converged if the absolute value of the directional derivative in the direction of the step is less than $10^{-8}$ or the difference in the cost function value between two consecutive iterations is less than $10^{-8}$. When running ALS we approximated the directional derivative as:

$$
\frac{E(U + \Delta U, V + \Delta V) - E(U, V)}{\sqrt{\|\Delta U\|_F^2 + \|\Delta V\|_F^2}}
$$

(78)

Where $U$ and $V$ are the values in the beginning of the iteration and $\Delta U$ and $\Delta V$ are the changes that were performed by the algorithm in the iteration.

Tab. 3 shows the number of iterations that each method performed before converging as well as the percentage of runs that converged before reaching the maximum number of iterations. The statistics are computed only over the converged runs. Tab. 4 and Tab. 5 show the statistics for the total runtime and the time per iteration. The experiment was performed on a PC with 2.66GHz Intel Core 2 Quad processor using MATLAB R2010b in single threaded mode. The code is available in the source code package that is part of this thesis. The implementations are not tuned for maximum performance and these numbers should not be taken as a measure of the potential speed of the algorithms.

The results show that Wiberg’s algorithm converges in a significantly lower number of iterations than LM and ALS. Also it converged in all runs while ALS converged only in 60% of the cases although it had an increased iteration limit (3000 instead of 300). Because of the lower number of iterations that are necessary, Wiberg’s algorithm converges faster, although its iterations times are approximately 2.5 times longer than LM and 30 times longer than ALS. Fig. 8.2 shows a cumulative histogram of validation error of the final solution. Each point on this plot show the number of runs that converged to a point with validation error less than or equal to the value on the x-axis. Wiberg’s algorithm clearly outperforms LM and ALS. It converged to the global minimum in over 140 out of 200 runs, while LM did the same in approximately 20 out of 200. ALS never reached the global minimum but a few runs reached values that are close to it.

Comparison of the performance of Wiberg’s algorithm with different values of the regularization parameter $\lambda$ is presented in Fig. 13(a). Results show that if a suitable weight is chosen for the regularization parameter, this can increase the frequency with which the algorithm converges to the global minimum. Also increasing the weight of the orthogonality constraints decreases the average number of iterations and the average running time of the algorithm (Tab. 3 and Tab. 4). Applying the same constraints with LM optimization algorithm also lead to a decrease in the number of iterations and the runtime.
<table>
<thead>
<tr>
<th></th>
<th>λ</th>
<th>mean</th>
<th>median</th>
<th>min</th>
<th>max</th>
<th>converged</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALS</td>
<td>0</td>
<td>1522.57</td>
<td>1503</td>
<td>96</td>
<td>2973</td>
<td>60.5%</td>
</tr>
<tr>
<td>LM</td>
<td>0</td>
<td>166.362</td>
<td>180</td>
<td>11</td>
<td>299</td>
<td>92.5%</td>
</tr>
<tr>
<td>LM</td>
<td>1e-05</td>
<td>144.396</td>
<td>149</td>
<td>11</td>
<td>299</td>
<td>98.5%</td>
</tr>
<tr>
<td>LM</td>
<td>0.0001</td>
<td>137.75</td>
<td>145</td>
<td>11</td>
<td>287</td>
<td>98%</td>
</tr>
<tr>
<td>LM</td>
<td>0.001</td>
<td>131.434</td>
<td>132.5</td>
<td>11</td>
<td>300</td>
<td>98%</td>
</tr>
<tr>
<td>Wiberg</td>
<td>0</td>
<td>44.315</td>
<td>33</td>
<td>10</td>
<td>180</td>
<td>100%</td>
</tr>
<tr>
<td>Wiberg</td>
<td>1e-05</td>
<td>46.095</td>
<td>38</td>
<td>11</td>
<td>216</td>
<td>100%</td>
</tr>
<tr>
<td>Wiberg</td>
<td>0.0001</td>
<td>36.575</td>
<td>30</td>
<td>10</td>
<td>146</td>
<td>100%</td>
</tr>
<tr>
<td>Wiberg</td>
<td>0.001</td>
<td>32</td>
<td>25.5</td>
<td>9</td>
<td>221</td>
<td>100%</td>
</tr>
</tbody>
</table>

Table 3: Number of iterations until convergence.

<table>
<thead>
<tr>
<th></th>
<th>λ</th>
<th>mean</th>
<th>median</th>
<th>min</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALS</td>
<td>0</td>
<td>24.3761</td>
<td>28.7625</td>
<td>1.06374</td>
<td>36.8557</td>
</tr>
<tr>
<td>LM</td>
<td>0</td>
<td>22.5693</td>
<td>23.7509</td>
<td>0.815505</td>
<td>39.388</td>
</tr>
<tr>
<td>LM</td>
<td>1e-05</td>
<td>20.8813</td>
<td>21.5632</td>
<td>0.90971</td>
<td>43.467</td>
</tr>
<tr>
<td>LM</td>
<td>0.0001</td>
<td>20.0494</td>
<td>20.9168</td>
<td>0.91515</td>
<td>44.3537</td>
</tr>
<tr>
<td>LM</td>
<td>0.001</td>
<td>19.1811</td>
<td>19.4056</td>
<td>0.90793</td>
<td>44.3353</td>
</tr>
<tr>
<td>Wiberg</td>
<td>0</td>
<td>14.7841</td>
<td>11.0878</td>
<td>3.02743</td>
<td>60.5895</td>
</tr>
<tr>
<td>Wiberg</td>
<td>1e-05</td>
<td>15.3595</td>
<td>12.7793</td>
<td>3.38493</td>
<td>69.279</td>
</tr>
<tr>
<td>Wiberg</td>
<td>0.0001</td>
<td>12.1494</td>
<td>9.78592</td>
<td>3.04298</td>
<td>47.7387</td>
</tr>
<tr>
<td>Wiberg</td>
<td>0.001</td>
<td>10.6074</td>
<td>8.28384</td>
<td>2.69137</td>
<td>72.5056</td>
</tr>
</tbody>
</table>

Table 4: Running time (seconds).

<table>
<thead>
<tr>
<th></th>
<th>λ</th>
<th>mean</th>
<th>median</th>
<th>min</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALS</td>
<td>0</td>
<td>0.0115673</td>
<td>0.0115539</td>
<td>0.0110785</td>
<td>0.0123617</td>
</tr>
<tr>
<td>LM</td>
<td>0</td>
<td>0.124534</td>
<td>0.127875</td>
<td>0.0741368</td>
<td>0.13105</td>
</tr>
<tr>
<td>LM</td>
<td>1e-05</td>
<td>0.140217</td>
<td>0.142091</td>
<td>0.0827009</td>
<td>0.146441</td>
</tr>
<tr>
<td>LM</td>
<td>0.0001</td>
<td>0.138568</td>
<td>0.141998</td>
<td>0.082865</td>
<td>0.148231</td>
</tr>
<tr>
<td>LM</td>
<td>0.001</td>
<td>0.137839</td>
<td>0.142162</td>
<td>0.0814647</td>
<td>0.150932</td>
</tr>
<tr>
<td>Wiberg</td>
<td>0</td>
<td>0.329286</td>
<td>0.329186</td>
<td>0.302743</td>
<td>0.363601</td>
</tr>
<tr>
<td>Wiberg</td>
<td>1e-05</td>
<td>0.331168</td>
<td>0.332082</td>
<td>0.307721</td>
<td>0.361889</td>
</tr>
<tr>
<td>Wiberg</td>
<td>0.0001</td>
<td>0.32942</td>
<td>0.329366</td>
<td>0.304298</td>
<td>0.357182</td>
</tr>
<tr>
<td>Wiberg</td>
<td>0.001</td>
<td>0.327501</td>
<td>0.327457</td>
<td>0.299041</td>
<td>0.368501</td>
</tr>
</tbody>
</table>

Table 5: Time per iteration (seconds).
Figure 12: Comparison of SFM algorithms on real-world data.
Figure 13: Effects of regularization
9 Conclusion

In this thesis we compared several popular algorithms for low-rank matrix factorization with missing data. The comparison was performed on synthetic low-rank matrices, synthetic and real-world structure-from-motion data. Results showed that Wiberg’s algorithm is able to solve problems with lower number of observed elements than the ones that can be solved by applying Levenberg-Marquardt or Alternating Least Squares. Also WA converges more often to the global minimum and in significantly less iterations than LM or ALS.

We improved Wiberg’s algorithm in two important ways. First, we demonstrated how the sparsity and the algebraic properties of the problem can be used to obtain a more efficient implementation. The per-iteration computational complexity of such implementation is similar to the most efficient implementations of generic non-linear least squares methods like Levenberg-Marquardt, when they are applied to the low-rank factorization problem. Experiments showed these advantages can be observed in practice as a reduction in the convergence times and the number of runs that get stuck in a local minimum.

Another improvement that we suggested is the addition of camera orthogonality constraints in a way that does not change the least-squares form of the cost function. Our results show that such constraints can further improve the convergence speed of WA and decrease the number of runs which get stuck in a local minimum.

These improvements make Wiberg’s algorithm practical to use on large problems and we believe it should be preferred over LM and ALS when solving low-rank matrix factorization or affine SFM problems. One exception are problems in which very large fraction of the matrix elements is observed. There ALS has a speed advantage due to its low iteration times. Because of the iterative nature of WA and ALS it is possible to combine the two algorithms in a way similar to the hybrid methods that Buchanan described in [1]. This can possibly result in a method that is fast on matrices with few missing elements but is also able to recover matrices with many missing entries. Another interesting path for future work is to explore the interactions between Wiberg’s algorithm outer iterations and the LSQR inner iterations. For example a way to dynamically control the stopping conditions of the inner iterations for different outer iterations may increase the speed of the algorithm by calculating the step direction with lower precision when that will be sufficient for the outer iteration.
References


