Implementing and Optimizing Regression with Linear Factored Functions

Georg Hieronimus
Technische Universität Berlin
georg.n.hieronimus@campus.tu-berlin.de

Zoltan Lux
Technische Universität Berlin
luxzolilux@hotmail.com

Supervisor: Phd. Wendelin Böhmer
Technische Universität Berlin
wendelin@ni.tu-berlin.de

Abstract

Linear Factored Functions (LFF) have the potential to break the curse of dimensionality by evaluating certain integrals analytically instead of sampling. A novel regression algorithm using LFFs has promising applications in Reinforcement Learning and Bayesian Inference.

To allow the wide spread use of LFFs this project presents an optimized implementation in C++ for the compression, regression and density estimation using LFFs. Especially the important compression algorithm was analysed in detail and the provided implementations outperforms Matlab in nearly all cases. In addition a fast solution for the problematic Memory consumption of this algorithm is presented. The provided implementation of the regression and density algorithm perform similar to MATLAB for big inputs and are faster by an order of magnitude for small inputs.
## Contents

1 Introduction 3
   1.1 Motivation .......................................................... 3
   1.2 Linear Factored Functions ......................................... 3
   1.3 Compressing Linear Factored Functions ......................... 3
   1.4 Regression with Linear Factored Functions ...................... 4

2 Implementation 6
   2.1 Selecting Math Libraries .......................................... 6
   2.2 Solving Linear Equation Systems .................................. 7
   2.3 Problems .............................................................. 8
      2.3.1 Numerical differences between MKL and Eigen ............ 8
      2.3.2 Error handling of Eigen / Mex ............................... 8

3 Testing 10
   3.1 Compression .......................................................... 10
      3.1.1 Creating compressible LFFs ................................. 10
      3.1.2 Runtime analysis: overview .................................. 11
      3.1.3 Runtime analysis: function norm calculation ............... 11
      3.1.4 Runtime analysis: loop iterations ........................... 16
   3.2 Regression ........................................................... 19
      3.2.1 Consistency ...................................................... 20
      3.2.2 Runtime .......................................................... 22
   3.3 Density Estimation .................................................. 25
      3.3.1 Scaling with the input parameters ........................... 25
      3.3.2 Accuracy ......................................................... 27

4 Conclusion 29
   4.1 Compression .......................................................... 29
   4.2 Regression ............................................................ 29
   4.3 Density Estimation ................................................... 29

A Appendix 31
1 Introduction

The section Introduction was written by Zoltan Lux.

1.1 Motivation

If we want to evaluate integrals over a Neural Network or Kernel functions, we have to sample the entire input space. However, integrals can be solved analytically for Linear Factored Functions. Analytical point-wise products and marginalization are beneficial structural properties of Linear Factored Functions.

1.2 Linear Factored Functions

The following section (based on [4]) provides a definition of Linear Factored Functions and a brief overview of some of their intriguing properties. Let \( \vartheta \) be the uniform distribution over \( \mathbb{Z} \subset \mathbb{R}^d \). A factored function \( g \in L^2 (\mathbb{Z}, \vartheta) \) is defined as the product of \( d \) one-dimensional Factor-Functions \( g^\alpha \in L^2 (\mathbb{Z}_\alpha, \vartheta^\alpha) \). Definitions over other distributions, like the diagonal Gaussian, are possible too. Due to Fubini's theorem, inner products of two factored functions can be written as the product of the inner product of the factored functions for each dimension:

\[
\langle g, h \rangle_\vartheta = \prod_{\alpha=1}^{d} \langle g^\alpha, h^\alpha \rangle_{\vartheta^\alpha}
\]

So the complexity of estimating the above integral by sampling drops from exponential to linear in the number of dimensions \( d \).

Linear Factored Functions (as defined in [5]) \( f \in F^m \) are a linear combination of \( m \) Factored basis functions.

\[
f(\vec{x}) := \vec{a}^T \Psi(\vec{x}) = \vec{a}^T \left( \prod_{\alpha=1}^{d} \Psi^\alpha(x_\alpha) \right) := \sum_{i=1}^{m} a_i \prod_{\alpha=1}^{d} \sum_{j=1}^{m_\alpha} B_{ji}^\alpha \phi_\alpha^j(x_\alpha)
\]

where \( a \in \mathbb{R}^m \) with \( m \) factored basis functions \( \psi \) with parameters \( B^\alpha \in \mathbb{R}^{m_\alpha \times m} \) for dimension \( \alpha \). Although only a small subset of functions can be expressed as factored functions, however in the limit of infinitely many basis functions Linear Factored Functions can approximate any function, where the basis functions are the Fourier bases. Fubini’s theorem also applies to Linear Factored Functions, which gives them the potential to break the curse of dimensionality. In the following two subsections we explain algorithms for regression with Linear Factored Functions, and then to compress Linear Factored Functions.

1.3 Compressing Linear Factored Functions

Point-wise addition and multiplication can increase the size of Linear Factored Functions by orders of magnitude, and also redundantly, as many of the basis are very similar. When finding more compact representations by the compression algorithm, the number of basis functions can be greatly increased while the difference in the learned function is negligible \( (< \epsilon) \). More formally, we have \( f \in F^m \), which we want to compress to \( \hat{f} \in F^\hat{m} \), where \( \| f - \hat{f} \|^2_0 < \epsilon \) and \( m \gg \hat{m} \). The greedy compression algorithm builds up \( \hat{f} \in F^\hat{m} \) incrementally, until the approximation error falls below \( \epsilon \). The compression follows a nested optimization and uses coordinate descend. Coordinate descend
updates one dimension at a time, which is a convex optimization problem, and has an analytical solution. The cost function in the inner loop from [4]:

$$\inf_{g \in \mathcal{F}} \| f - \tilde{f} - g \|_2^2 \quad \text{s.t.} \quad \| g^\alpha \|_\vartheta = 1, \quad \forall \alpha \in \{1, \ldots, d\}$$

In the inner loop the coefficients of the dimensions ($b^\alpha$) of the factored basis functions are updated. In the outer loop the $a$ coefficients of the new LFF are calculated by solving an ordinary least squares problem. The outer loop is repeated until the the original function $f$ is approximated sufficiently ($\| f - \tilde{f} \|_\vartheta^2 < \epsilon$).

Compression algorithm pseudocode from [4]

Algorithm 6 - greedy LFF compression, e.g. after a point-wise multiplication

| input: $B^\alpha \in \mathbb{R}^{m^\alpha \times m}, \forall \alpha, \ a \in \mathbb{R}^m$  // input function $f$
| $\hat{B}^\alpha := \emptyset, \ \forall \alpha; \ \hat{a} := \emptyset; \ \text{err} := \infty$  // initialize output $\tilde{f}$
| while $\text{err} > \epsilon$ do
| $c^\alpha := 1 \in \mathbb{R}^{m^\alpha + |\alpha|}; \ d_\alpha := \infty, \ \forall \alpha$  // initialize coefficients
| while $\max\{d_\alpha\} > \epsilon$ do
| for $\alpha$ in randperm($1, \ldots, d$) do
| $b_{uc} := [B^\alpha, \hat{B}^\alpha] \left( \prod_{\alpha} c^\beta \cdot \begin{bmatrix} a \\ -\hat{a} \end{bmatrix} \right)$  // unconstrained solution for $g^\alpha$
| $d_\alpha := 2 b_{uc} C^\alpha b^\alpha - d_\alpha$  // $d_\alpha = \| f - g^\alpha \|^2_\vartheta - \| g^\alpha \|^2_\vartheta - \| f \|^2_\vartheta$
| $b^\alpha := b_{uc} / \sqrt{b_{uc}^\top C^\alpha b_{uc}}$  // normalize functions to length 1
| $d_\alpha := 2 b_{uc} C^\alpha b^\alpha - d_\alpha$  // $d_\alpha = \| f - g^\alpha \|^2_\vartheta - \| f - g^\alpha \|^2_\vartheta$
| $\tilde{c}^\alpha := [B^\alpha, \hat{B}^\alpha]^\top C^\alpha b^\alpha$  // recompute coefficients
| end for
| end while
| $\hat{B}^\alpha := [\hat{B}^\alpha, b^\alpha], \ \forall \alpha; \ \hat{a} := (\prod_{\alpha} \hat{B}^\alpha)^\top C^\alpha (B^\alpha, \hat{B}^\alpha)^{-1} (\prod_{\alpha} \hat{B}^\alpha)^\top C^\alpha B^\alpha a$  // update $\tilde{f}$
| $\text{err} := \left[ \begin{bmatrix} a \\ -\hat{a} \end{bmatrix}^\top (\prod_{\alpha} [B^\alpha, \hat{B}^\alpha]^\top C^\alpha [B^\alpha, \hat{B}^\alpha]) \begin{bmatrix} a \\ -\hat{a} \end{bmatrix} \right]$  // error $\text{err} = \| f - \tilde{f} \|^2_\vartheta$
| end while
| output: $\hat{B}^\alpha \in \mathbb{R}^{m^\alpha \times \hat{m}}, \ \forall \alpha, \ \hat{a} \in \mathbb{R}^{\hat{m}}$

1.4 Regression with Linear Factored Functions

In the regression algorithm the cost function is measured with respect to the training distribution $\xi$ instead of the uniform distribution $\vartheta$. As $\xi$ usually samples the input space unevenly, the values of the the $y$ values of the learned function must be regularized in regions without samples. Virtual samples are assumed from the training set according to Equation 3.18 on Page 31 of [4]. The virtual samples are drawn from Gaussian distributions centered around the training set, so regions in the input space with many training samples receive many virtual samples. The virtual variance is then scaled by an uncertainty measure, thus smooth functions are enforced by regularization only in regions with little or no training data.
Regression algorithm pseudocode from [4]:

**Algorithm 8 – regression with LFF**

<table>
<thead>
<tr>
<th>Input</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$X \in \mathbb{R}^{n \times m}$, $y \in \mathbb{R}^n$, $\sigma^2 \in \mathbb{R}^d$ $\epsilon \in \mathbb{R}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$C_x := (\phi^\alpha, \phi^\beta)<em>{\alpha, \beta}$, $C</em>\alpha := (\nabla \phi^\alpha, \nabla \phi^\beta)_{\alpha, \beta}$, $\forall \alpha$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\phi_\alpha := \phi_\alpha(X_{\alpha \times \epsilon})$, $\forall \alpha$, $\forall j$, $\forall i$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$f := 0 \in \mathbb{R}^n$; $a := 0$; $B^\alpha := 0$, $\forall \alpha$; $\Psi := \infty$; $m := 0$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

while $det\left(\frac{1}{n} \Psi \Psi^T\right) > \epsilon$ do

  $b^\alpha := 1^\alpha \in \mathbb{R}^{m_\alpha}$, $\forall \alpha$; $g^\alpha := 1 \in \mathbb{R}^n$, $\forall \alpha$  

  $\alpha^\alpha := 1 \in \mathbb{R}^{m_\alpha}$, $\forall \alpha$; $\rho^\alpha := 1 \in \mathbb{R}^n$, $\forall \alpha$  

  $h_\alpha := \infty \in \mathbb{R}^d$; $d := 0 \in \mathbb{R}^d$  

while $\max_{\alpha_\alpha} \{h_\alpha\} > \epsilon$ do

  $b' := b^\alpha$  

  $C := \phi^\alpha \Phi^\alpha \Phi^\alpha + \sigma^2 \tilde{C}_\alpha + \sum_{\beta \neq \alpha} \sigma^2_\beta \tilde{C}_\beta$  

  $u := \phi^\alpha \left[ (y - \bar{f}) \prod_{\beta \neq \alpha} g^\beta \right]$  

  $u := u + \sigma^2 \tilde{C}_\alpha \Phi^\alpha a \cdot c^\alpha + C^\alpha B^\alpha \left( a \cdot \sum_{\gamma \neq \alpha} \sigma^2_\gamma c^\gamma \prod_{\beta \neq \gamma} c^\beta \right)$  

  $b_{\text{new}} := C^{-1} u$; $b^\alpha := b_{\text{new}} / \sqrt{b_{\text{new}} \Phi^\alpha b_{\text{new}}}$  

  $h_\alpha := 2(b^\alpha - b')^T C b_{\text{new}} + b'^T C b^\alpha - b^\alpha^T C b^\alpha$  

  $\alpha^\alpha := B^\alpha \Phi^\alpha b^\alpha$; $\rho^\alpha := B^\alpha \Phi^\alpha b^\alpha$  

  $g^\alpha := \Phi^\alpha \Phi^\alpha b^\alpha$; $d^\alpha := b^\alpha^T C b^\alpha$  

  for $\alpha \in \arg\min_{\alpha} \{h_\alpha\}$ do

  $C := \Phi^\alpha \Phi^\alpha \Phi^\alpha + \sigma^2 \tilde{C}_\alpha + \sum_{\beta \neq \alpha} \sigma^2_\beta \tilde{C}_\beta$  

  $u := \phi^\alpha \left[ (y - \bar{f}) \prod_{\beta \neq \alpha} g^\beta \right]$  

  $u := u + \sigma^2 \tilde{C}_\alpha \Phi^\alpha a \cdot c^\alpha + C^\alpha B^\alpha \left( a \cdot \sum_{\gamma \neq \alpha} \sigma^2_\gamma c^\gamma \prod_{\beta \neq \gamma} c^\beta \right)$  

  $b_{\text{new}} := C^{-1} u$; $b^\alpha := b_{\text{new}} / \sqrt{b_{\text{new}} \Phi^\alpha b_{\text{new}}}$  

  $h_\alpha := 2(b^\alpha - b')^T C b_{\text{new}} + b'^T C b^\alpha - b^\alpha^T C b^\alpha$  

  $\alpha^\alpha := B^\alpha \Phi^\alpha b^\alpha$; $\rho^\alpha := B^\alpha \Phi^\alpha b^\alpha$  

  $g^\alpha := \Phi^\alpha \Phi^\alpha b^\alpha$; $d^\alpha := b^\alpha^T C b^\alpha$  

  end for

end while

$B^\alpha := [B^\alpha, B^\alpha]$; $\Psi := \prod_{\alpha=1}^d \Phi^\alpha$; $m := m + 1$  

while $\sqrt{B^\alpha \Phi^\alpha B^\alpha} \neq \infty$ do

  $a := (\Phi \Psi \Psi \Phi)^{-1} \Psi g$; $f := \Psi^T a$  

end while

Output: $a \in \mathbb{R}^n$, $\{B^\alpha \in \mathbb{R}^{m_\alpha \times m_\alpha}\}_{\alpha=1}^d$  

The regression algorithm follows a nested optimization scheme using coordinate descend. In the outer loop we add new basis functions until the convergence criteria of the cost function is met. In the inner loop coordinate descent optimizes dimension $b^\alpha$ of new basis function $g = \prod_{\alpha} g^\alpha = \prod_{\alpha} b^\alpha \Phi^\alpha$, which is a convex optimization problem with an analytical solution. We can either iterate over a random permutation of all the input dimensions in the inner loop, or follow a heuristic. The cost function in the inner loop for parameter vector $b^\alpha$ of factored function $g^\alpha$ for dimension $\alpha$ is found on page 89 of [4]. As the optimization problem in the inner loop is convex, the solution can be obtained by setting the derivative to 0. When the optimization of $b$-s finished after adding the new factored basis, we calculate $a$ analytically by solving an ordinary least squares problem in the outer loop.
2 Implementation

This sections gives an overview over all decisions we made before starting the implementation and a review of the difficulties during the implementation.

*It was written by Georg Hieronimus.*

2.1 Selecting Math Libraries

The main aim of this project is to create a faster implementation of the compression and regression algorithm for LLF the the Matlab implementation provided by Wendelin. The new implementations should blend in easily in the Matlab Framework created by Wendelin.

The regression and compression algorithm consists both of two loops: one nested into the other (For a detailed explanation of the algorithms see Section 1). Matlab is known for its performance at matrix operations, but creates only slow loop implementations due to its just-in-time Compiler. Therefore an implementation in C or C++ (which compile to native code and thus create faster loops) should be faster then Matlab, if all matrix operations would be at least as fast as in Matlab.

Another area to explore is the possibility to utilise a GPU for the algorithms. The provided Matlab implementation works solely on the CPU. Two options were examined in this case: Using the Matlab-GPU support and an C++ implementation using CUDA (the GPU interface by Nvidia [2]).

To match the speed of Matlab at matrix operations a suitable Maths library has to be chosen. The library has to be robust and provide an algorithm for solving linear equations. In addition it has to work well with the Matlab-Mex compiler to blend into the current LFF Framework. We examined the following candidates: Eigen, Blaze and Armadillo.

All three libraries can be used as a wrapper around a given BLAS library. BLAS (Basic Linear Algebra Subprograms) is a specification for a set of common low-level linear-algebra operations. The most famous implementations of the BLAS interface are OpenBLAS and MKL (created by Intel and optimized for Intel processors, freeware [3]). Beside its usage as wrapper around a BLAS library, Eigen provides its own implementation for its algorithms.

Nvidia provides a BLAS implementation operating on the GPU with the CUDA libraries: cuBLAS [1]. Other possibilities were not tested.

It wasn’t possible to get Armadillo work with the Matlab-Mex compiler, therefore only Eigen and Blaze were compared regarding their speed.

To evaluate the performance the most interesting aspect of the library is the speed of its matrix multiplication. Figure 1 a shows a comparison between Eigen, Blaze, Matlab, Matlab using the GPU and CUDA.

Eigen’s own implementation is noticeable slower then all other options. Using MKL or OpenBLAS, Blaze and Eigen are nearly as fast as Matlab. Which was expected, as Matlab uses a special MKL version internally. The small time difference is not significant but could be interpreted as the overhead induced by using Mex. Both implementations using the GPU are slower then the MKL libraries. The time differences are probably mainly induced by the transfer to the GPU, as Nvidia claims its cuBlas library to be faster then MKL [1]. Using the GPU only for some multiplications seems therefore not beneficial, only a complete implementation of the algorithm on the GPU might perform faster.

Due to its reputation and the small differences in the multiplication time (even though its not significant), we decided to use Eigen with MKL.
(a) Comparing C++ libraries and Matlab. For the Matlab (GPU) and CUDA implementation the
time to transfer the data from the CPU to GPU and back was included into the measurement. The differences
between Matlab, Eigen (MKL), Blaze(MKL / OpenBLAS) are not significant, the errorbars of all four
lines overlap.

Figure 1: Execution time of a matrix by matrix multiplication over the size of the matrix. The
matrices were generated randomly by the Matlab uniform number generator rand. Each matrix size
was tested 50 times, the provided points are the mean value of these tests. The errorbars had to be
omitted to give the reader a clear image of the trends. The timing was analysed using the same test
system as used for the compression algorithm. See Section 3.1.2 for details.

(b) Eigen with MKL and different number of Threads vs Matlab

Multithreading with MKL  MKL can use a different number of threads. At most it uses the
physical number of cores of the current system (4 on the test system). Whereas some libraries have
problems utilizing all cores and can show (in rare cases) worse performance with a higher number of
cores, MKL doesn’t have any problems to use more threads for its advantage as Figure 1 b shows.
Therefore all computations were carried out with maximum number of threads.

2.2 Solving Linear Equation Systems

The regression and compression algorithm need to solve systems of linear equations. Eigen does
provide a few different algorithms for this tasks. Most of them work for all types of matrices, but LLT
works only with positive definite matrices. In both algorithm only positive matrices are expected.
Figure 2 a shows the speed of all algorithm on positive definite matrices (Figure 27 in the Appendix
shows the speed for all types of matrices.).

Comparing the the speed of Matlab for positive matrices and general matrices indicates that Matlab
tests its input for this case and uses a different and faster algorithm for positive matrices. Nevertheless,
in both cases provides Eigen a slightly faster algorithm namely: LLT for positive matrices and
HouseholderQr for general matrices. Beside the speed of the algorithms, the error was analysed by
calculating $A \cdot X - B$. The mean of this error for positive definite matrices is displayed in Figure 2 b
(Figure 28 in the Appendix shows the error for all types of matrices.).

The error is quite high in the case of positive definite algorithms, but equally high for all algorithms.
All in all provides Eigen with LLT a algorithm for positive definite matrices, which is faster then
Matlab and produces equally well results. This algorithm is used in both implementations.
Figure 2: A is positive definite and was generated randomly by the Matlab uniform number generator rand. Each matrix size was tested 50 times, the provided lines display the mean value of these tests. Eigen was compiled with MKL. The timing and error were analysed using the same test system as used for the compression algorithm. See Section 3.1.2 for details.

2.3 Problems

The implementation of regression and compression took far longer then expected. There are two main reason for this:

2.3.1 Numerical differences between MKL and Eigen

Matlab and Eigen using MKL produce exactly the same results for Matrix multiplication between two equal sized matrices. But this is nearly the only case. Matrix times vector multiplication results in a mean difference around $10^{-13}$ per matrix multiplication, solving linear equations produces slightly different results (as seen in Figure 2 b) and even the calculation of a square root results in small distinctions.

In addition, the order of operations can induce a noticeable difference. For example squaring all elements of a matrix and then calculating the product of its rows:

$$A = A \cdot ^2;$$
$$v = \text{prod}(A);$$

Performing the multiplication for the product first increases the performance by reducing the number of necessary multiplications and therefore reduces the numerical error induced by these. All these small variations increase over all the iteration of the algorithms and result in differences between the results even though the code is performing exactly the same operations.

2.3.2 Error handling of Eigen / Mex

It is very complicated to find bugs in Code compiled with Matlab. It is not possible to attach a debugger to code called by Matlab. Matlab crashes immediately if any unexpected read or write is performed and especially if the returning values of the mex-programm are flawed in any way. A crash produces a stacktrace, but without any hint in which part of the code the error occurred.

In addition Eigen does not create any errors if the size of matrices involved in multiplication doesn’t match. The matrices are silently expanded and the program crashes on a later stage when the expanded
matrices are saved or otherwise used. especially bad is this behaviour for multiplication between elements of two matrices. If the matrices are not matching, Eigen will produce MKL calls which are not conform to the API. MKL will then use memory around the given matrices and therefore sometimes produce NaN values (crashing the program) and sometimes just small values resulting in runs without any error.

All in all this results in a very painful and time consuming debugging process mostly using print-statements to find errors.
3 Testing

All three implementations were tested regarding their speed and accuracy compared to Matlab.

*The subsection about compression and the subsection about density estimation were written by Georg Hieronimus, the subsection about regression was written by Zoltan Lux. The implementations of all algorithms was written by Georg Hieronimus.*

3.1 Compression

The compression algorithms purpose is to reduce the number of factored functions $m$ of a given LFF (linear factored function). The algorithm consists of roughly two parts:

1. Calculating the function norm of the input LFF.
2. Creating a new LFF iteratively, by calculating a new factored function and adding it to the current solution until the error between the current function norm and the input falls below a threshold.

The second part is carried out by two nested loops: The outer loops performs until the error is below a threshold and the inner loop calculates an new optimal factored function using coordinated descent. All mentions of inner loop iterations always refer to the total number of inner loop iterations. For details of the algorithm and its theoretical background see section 1.3.

The input of the compression algorithm can vary in three dimensions: The number of factored functions $m$ of the given function, the number of dimensions $d$ and the number of basis functions $b$.

For the current implementation, the basis function is assumed to be a Fourier function.

3.1.1 Creating compressible LFFs

To inspect the runtime of the algorithm over the three input dimension $m, d, b$ suitable input LFFs have to be created. Internally, one LFF is represented by two variables:

- A Vector of size $m$ called $a$ with the linear coefficients of all factored functions.
- A list of matrices called $B$, one for each dimension $d$. Each matrix of this list contains the linear coefficients for each basis function of all factored functions and has therefore a size of $b$ by $m$.

To generate a compressible LFF with given $m, d$ and $b$ the following steps are performed:

1. $b$ coefficients of one basis function are drawn randomly from a uniform distribution using the Matlab pseudo-random number generator rand.
2. The generated coefficients are scaled exponential by multiplying with $\exp(-[1:6])$\(^1\) and afterwards normalized by dividing each coefficients by the sum of all coefficients.
3. The vector of coefficients is copied $m$ times to create one of the Matrices of $B$.
4. Step 1-3 are repeated for each dimension $d$ resulting in the list of Matrices $B$.
5. The coefficient vector $a$ is created by drawing randomly using rand and then scaled by multiplying with $\exp(-[1:6])$.

\(^1\) $1:6$ is more or less arbitrary, one could use $1:5$ or $1:7$ instead. I wanted to restrict the scaling to prevent the last coefficients to become meaningless, therefore a fixed part of the exponential function was used.
The exponential scaling is performed to imitate the look of a LFF created by regression: Normally higher order basis functions have a low coefficient, because they represent strong oscillating functions. This has probably no effect on the performance measurements.

The created LFF is trivially compressed by using only the first factored functions and setting its coefficient to the sum of \( a \). The compression algorithm is able to find this or a very small solution in most of the cases. This results in a very short execution time with very few loop iterations (typically around 20 inner loop iterations). The difficulty can be increased by adding a random noise to the \( B \) matrices, thus increasing the number of loop iterations. The algorithm is quite sensible to added noise and will often not find a better solution.

### 3.1.2 Runtime analysis: overview

It is necessary to know which part of the algorithm is executed most of the time to improve the runtime of the compression algorithm. An analysis of the Matlab code using the Matlab profiler provides the following insights:

1. Calculating the function norm of the input LFF takes far longer then one loop iteration. The artificially created compressible LFFs only need around 20 loop iteration to find the solution, the calculation of the function norm takes in these cases more then 90% of the execution time. This is especially valid for a higher number of factored functions \( m \).
2. During the loop iteration, three parts of the code take nearly equal amount of execution time.

The ratio between the time spend for calculating the function norm and time spend in the loop, depends naturally on the number of loops. To allow the reader to achieve a clear picture of the runtime scaling with the input parameters, both parts of the code are examined independently in the next two subsections.

All runtime tests for the compression algorithm were performed on a test system with the following properties:

- Processor: Intel Core i5-2400 CPU with 3.10 GHz clock frequency in 4 cores
- Graphics: GeForce GTX 960 with 4GB Ram
- RAM: 8 GB
- OS: Ubuntu 15.04 64-bit

### 3.1.3 Runtime analysis: function norm calculation

The first of the compression algorithm is the function norm calculation, Figure 3 shows the Matlab code for this calculation.

For this algorithm, a few performance optimizations are possible:

- \( fC \) is symmetric, because it is the covariance matrix of \( B\{d\} \). Therefore only the upper half of the matrix has to be calculated. Matlab normally detects this by itself. Using the corresponding algorithms for Eigen did not increase the computation speed.
- In the special case of \( d = 1 \), a much faster order of calculation is possible. In this case \( fC \) is not necessary and \( A^* B1' \) can be performed as the first calculation - effectively reducing the size of the matrix and the number of calculations. Matlab does not detect this case by itself, a special case handling has to be performed. The scaling in the case of \( d = 1 \) was not
Figure 3: The Matlab code for the function norm calculation. 99% of the execution time of this algorithm are used in line 6.

inspected further. It can not be excepted that the results presented in this subsection apply in this case.

- The covariance matrix is always an identity matrix and can therefore be omitted. This should reduce the computation time, especially for inputs with high number of Fourier bases $b$.

During the runtime analysis, both implementation were tested: a Matlab implementation with the covariance matrix and a Matlab implementation without a covariance matrix (simply called 'Matlab').

The memory consumption of this algorithm is approximately $m + m \cdot b + 2 \cdot m^2 = O(m^2 + m \cdot b)$. On the used test system this limits the maximum number of factored functions $m$ to around 2000 while using the CPU and to around 12000 while using the GPU. To circumvent this problem, an iterative calculation of $A^T \cdot fC$ can be performed: In each iteration only $k$ columns of $fC$ are calculated and used for the multiplication of $A^T \cdot fC$. This reduces the memory consumption to $O(m \cdot k + m \cdot b)$, but increases the calculation time. $K$ is chosen automatically by the algorithm to use all the remaining memory. For details see the implementation in the Appendix (Figure 29). In the following time analysis, the adapting algorithm is called AdaptMemory. Besides its implementation for the CPU in Matlab, the code was also adjusted to work with the GPU.

**Scaling with the input parameters**  The most important parameter for the compression algorithm is the number of factored functions $m$, because this values should be reduced and could therefore be assumed to be quiet high in the input. To give a more detailed impression of the scaling regarding $m$, the time data is presented in two Figures: Figure 4 shows the scaling with a small $m$ up to 1000 and Figure 5 shows the scaling with a bigger $m$.

**For small $m$:** the C++ implementation is faster then all other implementations for this input size (not significant). The AdaptMemory implementations and Matlab without the covariance matrix perform nearly as well. Matlab with covariance matrix performs worst. The GPU implementations are quiet slow for $m < 700$. Up to around $m = 200$, the Cuda implementation is the faster then the other GPU implementations, but still slower then C++. At $m = 1000$, the Matlab implementation using the GPU is the fastest implementation, followed by the AdaptMemory implementation on the GPU. The CPU implementations are all slower then the GPU implementations at this point.

**For $m$ bigger 1000:** the fastest implementation for the CPU are AdaptMemory and surprisingly the Matlab implementation with the covariance matrix. The Matlab implementation is the slowest, the
Figure 4: The calculation time of the function norm for the compression algorithm over the number of factored functions $m$. Displayed only up to $m = 1000$. Each parameter was tested 50 times, the lines display the mean of these tests. In each test 50 Fourier functions and 4 dimension were used.

Figure 5: The calculation time of the function norm for the compression algorithm over the number of factored functions $m$. Displayed for $m > 1000$. Each parameter was tested 20 times, the lines display the mean of these tests. In each test 50 Fourier functions and 4 dimension were used. Only the AdaptMemory implementation is able to handle inputs with bigger $m$ (indicated by continuing the plot until the right side), because of the RAM limitations of the test system. For the GPU implementation only 2GB of RAM are available, therefore stops their plot at 12000.
C++ implementation is in between. The reason for the speed increase through the covariance matrix is unclear. As mentioned, Matlab seems to detect if you multiply the same matrix by itself and changes the algorithm accordingly. It could be that the covariance prevents Matlab from detecting that the result will be symmetric and that the symmetric algorithm is slower then the normal for big matrices. Tests with all other implementations, especially with the AdaptMemory algorithm, did not show this speed increase.

The GPU implementations are always faster then the CPU implementations for \( m > 1000 \). Cuda is the slowest implementation in this case, AdaptMemory and Matlab are equally fast. A big deficit for all algorithms is the memory consumption. Only 2GB RAM are available on the GPU, therefore Cuda and Matlab (GPU) cant handle \( m < 12000 \). For the CPU versions, this value is increased to 20000, but the AdaptMemory implementation is easily able to solve instances with \( m > 50000 \).

Figure 6 shows the result for the scaling with the number of dimensions \( d \). All implementations display a approximately linear scaling with \( d \). For the calculation on the CPU: C++ is always 5-10% faster then Matlab (without covariance matrix). The AdaptMemory implementation is always as fast as C++. The calculation on the GPU scales better then on the CPU. The Cuda version performs worst, and the Matlab version performs best, but the difference between all 3 versions is very slim. All in all remains the relative difference between the implementation on the CPU / GPU the same, non of them scales better then the other.

![Factored Functions: 1000 Fourier-Basis: 100](image)

Figure 6: The calculation time of the function norm for the compression algorithm over the number of input dimensions. Each dimension was tested 30 times, the lines display the mean of these tests.

Figure 7 shows the result for the scaling with the number of Fourier bases \( b \). On the left is the scaling for the CPU implementations, on the right for the GPU. Matlab with covariance matrix scales worst. This was expected, because the additional Cov-matrix has size \( b \times b \). C++ and the AdaptMemory algorithm scale equally, while Matlab without covariance tops both of them. The GPU implementations scale all equal, but worse then the CPU implementations. The scaling between all
these implementations is far more different then the scaling with the number of factored functions (Notice: Figure 7 is a log-log plot, whereas Figure 5 is a linear plot.). For that reason dominates the number of Fourier-bases the difference between the runtime of the implementations in most cases. For example, tests showed that with \( b = 1000 \), up to the RAM limit of 20000 factored functions, the Matlab implementation (without covariance matrix) is faster than all other implementations (on GPU or on CPU). Even though Matlab without covariance matrix scales worse than all other implementations with the number of factored functions (See Figure 30 in the Appendix).

![Graphs showing runtime comparison](image)

Figure 7: The calculation time of the function norm for the compression algorithm over the number of Fourier dimensions. Each parameter was tested 30 times, the lines display the mean of these tests. 1000 factored functions and 4 dimensions were used in each test.

**Accuracy of the norm calculation algorithm** All implementations produce similar results and could therefore be The relative error\(^2\) between Matlab and the all other implementations was during all the tests below \(< 10^{-12}\).

**Conclusion** The number of dimensions has only a small influence on the choice which implementation to pick for a task, because all implementations scale linear with this parameter, even though the GPU implementations scale slightly better. In contrast, the algorithms scale quadratic with the number of factored functions and the number of Fourier bases. But the implementations scale quite similar

\[ \text{relative Error: } \frac{\text{abs}(\text{Norm}_{\text{Matlab}} - \text{Norm}_{\text{other}})}{\text{Norm}_{\text{Matlab}}} \]
with the number of factored functions. In contrast, the scaling with the number of Fourier bases differs strongly. Therefore should the decision which algorithm to use for a productive environment depend on the number of Fourier bases \( b \). For \( b < 200 \) provide the GPU implementations an enormous speedup. Due to the Memory limitations of the GPU, AdaptMemory (GPU) is then probably the best pick. If it is not possible to use the GPU, one should use AdaptMemory on the CPU or Matlab with covariance. For \( b > 1000 \) is Matlab without covariance matrix the best pick, if the RAM usage is not a problem. Otherwise AdaptMemory on the CPU is the best choice.

3.1.4 Runtime analysis: loop iterations

During the loop iteration, three parts of the code take nearly equal amount of execution time. Figure 8 shows the Matlab code of this lines.

```matlab
% Initialize all coefficients
for d = 1:NDIM
    coeff{d} = [Bbar{d} lff.B{d}]’ * gb{d}; % 20% of the time
end;

% Gradient of target function, i.e. \( <g,f> \)
  buc = [Bbar{d} lff.B{d}] * sc; % 38% of the time

% Update coefficients
coeff{d} = [Bbar{d} lff.B{d}]’ * gb{d}; % 36% of the time

Figure 8: The three most expensive lines in the inner loop calculation of the compression algorithm.

All three lines are matrix times vector multiplications and can probably not be improved. But the main assumption for this project remains: a C++ implementation should provide faster loop execution. In adding to the implementation in C++, a Matlab implementation using the GPU is provided.

For inputs with the same number of factored functions \( m \), number of input dimensions \( d \) and number of Fourier bases \( b \) may the number of iterations to find the perfect result still vary heavily. In this case the execution time varies strongly too. To achieve a clear picture of the scaling properties of the implementations, the total execution time is divided by the number of inner loop iterations. In addition, to ease the collection of data, the inner and outer loop iterations were capped at 10 iterations \(^3\). Thus resulting in a maximum of 100 loop iterations per tested configuration. Previous tests with higher number showed no time difference per iterations for higher caps.

Scaling with the input parameters Figure 9 displays the result for the number of factored functions \( m \). The C++ implementation is always faster then the provided Matlab implementation even though the relative difference between both declines. The Matlab implementation needs still twice the time of the C++ implementation at 50000 factored functions. The GPU implementation is the slowest implementation for small inputs \( m < 3000 \), but the runtime per loop doesn’t change much until \( m = 10000 \). For \( m > 15000 \) is the GPU implementation faster then C++, at \( m = 50000 \) is the GPU around 35% faster then C++.

\(^3\)called MAXCONVERGE / MAXAPPROX in the Code
Figure 9: The time per loop for the loop iterations of the compression algorithm measured for inputs with varying dimensions. Each configuration was tested 20 times.

Figure 10 displays the result for the number of Fourier bases $b$. Matlab scales better then C++, but C++ is faster then Matlab until $b > 20000$. For small $b (< 100)$ is C++ more then 10 times faster then Matlab.

The GPU implementation shows a similar development like with parameter $m$: For small values of $b$, it is the slowest implementation. But for $b > 600$ is the GPU faster then Matlab and for $b > 4000$ it is faster then C++.

At $b = 40000$ takes C++ $1.5 \times$ the time of Matlab and the GPU is $3 \times$ faster then Matlab.

Figure 11 displays the result for the number of input dimensions $d$. All three implementations scale equal with $d$, the relative difference remains the same: At $m = 500$ and $b = 100$ is C++ around 70% faster then Matlab. The GPU implementation is always the slowest implementation for this configuration.

**Accuracy of the loop iterations algorithm** The difference between Matlab and Eigen (and the GPU implementation) was tested by comparing the error of the created function in comparison with the input function. To calculate the error, the computed function and the input function was evaluated at 100 points equally spaced between 0 and 1 (for each input dimension). This error was calculated for all the previous shown test cases for the loop implementation and in addition for several tests of the complete algorithm. The error of all three implementations was always similar to each other. As an example, Figure 12 shows the difference while varying the number of dimensions.
Figure 10: The time per loop for the loop iterations of the compression algorithm measured for inputs with varying dimensions. Each configuration was tested 20 times.

Figure 11: The time per loop for the loop iterations of the compression algorithm measured for inputs with varying dimensions. Each configuration was tested 20 times.
Conclusion  For typical inputs with a high number of factored functions and a low number of Fourier bases ($b < 500$) and few input dimensions ($d < 20$) does the C++ implementation provide a huge performance gain compared to the previous Matlab implementation. Only in special cases with an enormous amount of Fourier bases does Matlab still outperform C++.

The GPU implementation is only useful if the size of the input reaches a certain hight. With a small input the transfer between CPU and GPU dominates the algorithm. But for big inputs especially above 15000 factored functions provides the GPU implementation a noticeable speed increase over the C++ implementation.

3.2 Regression

We tested regression on 4 different test functions, and aggregated the results on all 4 to produce the following plots. The individual points are the mean of 50 runs of over the 4 functions, where we also show the standard deviations for these $4 \times 50$ runs. The 4 test functions originally had 2 input dimensions, and for more than 2 dimension they could were calculated by taking the L2-Norm or the squared L2-Norm of the input variables for function 1,2 and 3. For the last (4th) approximated function only the first 2 input dimensions influenced the output ($y$), the others were simply noise. We added white Gaussian noise to the regressed labels, quantified as a Signal to Noise Ratio of 40dB per sample.

Function 1 is defined as:

$$y = \sin(\|\vec{x}\|)$$
Function 2 is defined as:

\[ y = - \sin \left( \|\vec{x}\| \right) / \|\vec{x}\| \]

Function 3 is defined as:

\[ y = \|\vec{x}\| \sin \left( \|\vec{x}\|^2 \right) / \|\vec{x}\|^2 \]

Function 4 is defined as:

\[ y = \sum_{k=0}^{20} a_k \cos \left( 2\pi \cdot x_k^2 \cdot x_1 \right) \]

Function 4 is the 1st 20 components of the Weierstrass function. The Weierstrass function is defined as:

\[ w(x) = \sum_{k=0}^{\infty} a_k \cos (2\pi b^k x) \]

where we interpreted \(x\) as \(x_1\) (1st input dimension) and \(b\) as \(x_2\) (2nd input dimension). The other dimensions of \(\vec{x}\) did not influence the output. An interesting property of the Weierstrass function is, that it is continuous everywhere but differentiable nowhere.

![Figure 13](image1.png)

Figure 13: \(y_1 = \sin \left( \|\vec{x}\| \right)\), \(y_2 = - \sin \left( \|\vec{x}\| \right) / \|\vec{x}\|\), \(y_3 = \|\vec{x}\| \sin \left( \|\vec{x}\|^2 \right) / \|\vec{x}\|^2\), \(y_4 = \sum_{k=0}^{20} a_k \cos \left( 2\pi \cdot x_k^2 \cdot x_1 \right)\)

The 4 approximated functions for 2 input dimensions with a signal to noise ratio of 40 dB on the target variable (y).

The input vectors of the training and test sets were drawn from the i.i.d uniform distribution, \(\vec{x} \sim U (-3, 3)^d\). \(y\) is always the target variable and \(\vec{x}\) is the input vector. The regularization was always identical for all input dimensions (\(\vec{\sigma}\)). For each measurements of a plot the regularization was constant, as changing \(\vec{\sigma}\) also influences the runtime, as larger values result in faster convergence.

### 3.2.1 Consistency

We also measured the training and test errors (MSE) to the approximated functions along with the difference (MSE) between the predictions of MATLAB and C++.

The training and test errors to the approximated functions evolve virtually identically for both MATLAB and C++. The difference in the individual training and test errors MATLAB and C++ is close to the numerical precision of calculations, and cannot be seen by the naked eye on our plot (Figure 14). The training error was calculated for each run individually, and was then aggregated. The training and test sets were both drawn from an i.i.d uniform distribution from \([-3, 3]^d\) in each input dimension. The number of learned Factored Functions was always the same for MATLAB and C++.

In Figure 14 Consistency means the Mean Squared Error measured between the reference MATLAB version and the C++ version over the test set. The test set was identical for all runs of the tests for the same number of dimensions and had no label noise. The label noise (\(y\) values) was removed from the test set to enable more accurate measurement of the generalization error. The difference between the predictions of the MATLAB and C++ versions is almost always close to the numerical precision of the calculations. There is only one case, where we had a noticeable difference (for 30 dimensions)
due to a single run, but on average it is still an order of magnitude smaller than both the training and test error of both implementations. The training error of MATLAB and C++ along with the test errors is practically identical, as seen shown on Figure 14.

Figure 14: Mean Squared Error measures of Regression for the tested number of input dimensions of the MATLAB and C++ LFF Regression algorithm implementations. The results were aggregated over the 4 tested functions for 3000 data points, 20 Fourier bases per input dimension, regularization $\vec{\sigma} = 2 \cdot 10^{-1}$ and Signal to Noise Ratio of 40 dB for the target variable ($y$). The test sets of the 4 functions were constant for all the runs and had no label noise. The averaged Mean Squared Error measures were calculated over all the 4 test functions, and the runtime for each function in each dimension was measured 50 times. Both the training and test errors of MATLAB and C++ appear practically identical.

The evolution of the test and training errors also demonstrates the robustness of LFF as the number of dimensions increase. The aggregated training and test errors stay more or less the same with some fluctuations above 5 dimensions over the tested numbers of dimensions.

Figure 15: The consistency of C++ to MATLAB is the Mean Squared Error between the predictions ($y$) of MATLAB and C++ on the validation set. The two implementations are in almost all cases practically consistent. For 30 input dimensions ($x$) we had a relatively smaller anomaly in a single case. In the rest of the runs the results in 30 dimensions were just as consistent as for different input dimensionality.
The consistency measure between C++ and MATLAB for most tested dimensions was close to the numerical precision of calculation with very little standard deviations. There was a single anomaly with a value of 0.548 for 30 input dimensions. Apart from this single occurrence, the results of C++ in 30 input dimensions were consistent to MATLAB as well.

### 3.2.2 Runtime

The C++ version was significantly faster than MATLAB for small inputs, due to the significantly higher cost of iterations in MATLAB (16). For large inputs MATLAB became faster. One should note, that for large inputs (e.g. in the number of training samples) C++ was not fully utilizing the CPU according to the configured 4 cores for MKL in contrast to MATLAB. The regularization for regression was relatively small ($\vec{\sigma} = 1 \cdot 10^{-4}$), due to the high number of training samples in most cases, which were distributed uniformly in the input space.

All runtime tests for the regression algorithm were performed on a test system with the following properties:

- Processor: Intel Core i7-4790 CPU with 3.60 GHz clock frequency in 4 cores with a total of 8 threads and 8 MB Cache
- GPU: GeForce GTX 960 with 2GB GDDR5 RAM
- Main Memory (RAM): 32 GB DDR3 1600MHz
- OS: Ubuntu 14.04 LTS 64-bit

![Number of Points and Runtime of Regression per Factored Function average speedup:2.2684](image)

Figure 16: Runtimes of the MATLAB and C++ LFF Regression algorithm implementations aggregated over the 4 tested functions for 2 input dimensions, 20 Fourier bases per input dimension, $\vec{\sigma} = 1 \cdot 10^{-4}$ and Signal to Noise Ratio of 40 dB for the target variable ($y$). The standard deviations and (mean) runtimes were calculated over all the 4 test functions, and the runtime for each function in each number of data points was measured 50 times.
Both the C++ and MATLAB implementations scale linearly in the number of training samples above a certain number of points.

The C++ implementation outperforms MATLAB in the number of dimensions. Although the fluctuation (measured by standard deviation) of the runtime is considerable, but it is still much smaller than the overall trend of the speedup for C++ in the number of dimensions. The LFF Regression implementations’ runtimes were aggregated over the 4 tested functions for 3000 data points, 20 Fourier bases per input dimension, regularization $\sigma = 2 \cdot 10^{-1}$ and a Signal to Noise Ratio of 40 dB for the target variable $(y)$. In this test case we needed to regularize more for high dimensional input spaces with the a relatively small number of samples. Even though high regularization was not needed for low dimensional training sets, it still had to be kept at the level needed for larger ones, because decreasing $\sigma$ results in slower convergence, therefore longer runtime.

One can observe an increasing speedup for C++ versus MATLAB in the number of dimensions, depicted on Figure 18. The growing performance gain can be explain by the increasing number of loop iterations performed by the coordinate descent optimization, which iterates over all dimensions sequentially in the inner loop of regression, and loops are faster in C++ than in MATLAB by an order of magnitude. Both implementations scale linearly in the number of dimensions over the numbers of input dimensions.

![Figure 17: Runtimes of the MATLAB and C++ LFF Regression algorithm implementations aggregated over the 4 tested functions for 3000 data points, 20 Fourier bases per input dimension, regularization $\sigma = 2 \cdot 10^{-1}$ and Signal to Noise Ratio of 40 dB for the target variable $(y)$. The standard deviations and (mean) runtimes were calculated over all the 4 test functions, and the runtime for each function in each dimension was measured 50 times.](image)
Figure 18: Speedup of C++ Regression compared to the original MATLAB implementation over the 4 tested functions for 3000 data points, 20 Fourier bases per input dimension, regularization $\vec{\sigma} = 2 \cdot 10^{-1}$ and Signal to Noise Ratio of 40 dB for the target variable ($y$).

We tested the runtime of both implementations for a varying number of Fourier basis as well (shown on Figure 19), which were the number of cosine basis for the factored basis functions. The more factored basis functions we take for the LFF, the bigger the model complexity can potentially grow. Therefore, one should expect an increasing runtime as the model complexity grows. We kept the regularization term $\vec{\sigma} = 2 \cdot 10^{-1}$, which is a relatively high value. The enforced smoothness of the learned function is needed because the number of Fourier bases is high compared to the number of samples (e.g. 100-200 Fourier bases per dimension for 3000 points). MATLAB outperforms C++ in the number of Fourier bases, however the standard deviations of the runtime is comparable to the complete runtime per learned factored function.

Figure 19: Runtimes of the MATLAB and C++ LFF Regression algorithm implementations aggregated over the 4 tested functions for 3000 data points, 2 input dimensions, relatively higher regularization $\vec{\sigma} = 2 \cdot 10^{-1}$ and Signal to Noise Ratio of 40 dB for the target variable ($y$). The standard deviations and (mean) runtimes were calculated over all the 4 test functions.
Conclusion of Regression Testing  The C++ implementation of LFF Regression is practically consistent to the original MATLAB version, which means that on average the difference in the target values of the learned functions of C++ and MATLAB are negligible over the test set compared to the training error (both are measured as mean squared errors). The C++ version is faster than MATLAB for small number of training samples and high dimensional input spaces. It is important to note, that the speedup factor grows in the number of input dimensions over the tested numbers of dimensions.

3.3 Density Estimation

The density estimation algorithm for the LFF is quite similar to the regression and compression algorithm: It consists of two loops nested into each other. The input for the density estimation is a list of data points (points \( p \times \text{dimension } d \)), and the number of Fourier bases \( b \) the algorithm should use for each factored function.

For the tests for this algorithm were only the spiral data set used, proposed by Wendelin [4, page 87]. Figure 20 shows an example of the used data and the estimation created by Matlab and C++.

![Figure 20: The density estimation of the spiral data. The images were created using 50 Fourier-Bases and 800 data points. There is no visual difference between Matlab and C++.

3.3.1 Scaling with the input parameters

The scaling of the runtime was tested for all three parameters: input dimension \( d \), number of data points \( p \) and number of Fourier bases \( b \). The number of inner loops was capped at 50 and the number of outer loop to 20. All tests were performed using a outerSigma and sigams of 0.0005. Due to variations in the number of loops for the same input parameters, only the runtime per inner loop is presented (the total runtime divided by the number of inner loops).

Figure 21 shows the scaling with the number of input Points. Matlab scales far better then C++ and is faster for \( p > 1000 \). For small inputs (\( p < 100 \)) is C++ more then 10 times faster then Matlab.

Figure 22 displays the scaling with the Fourier bases \( b \). For small number of bases is C++ faster then Matlab, but for \( b > 1000 \) are both implementation equally fast.

Figure 23 displays the scaling with the number of dimension \( d \). Matlab scales worse then C++ in this case. Both seem to scale linear with \( d \), but C++ with a lower slope then Matlab (For \( d > 10 \): A fit of the mean values estimates a slope of \( 4 \cdot 10^{-4} \) for Matlab and \( 7 \cdot 10^{-5} \) for C++.)

\(^4\)Wendelin you should check the equation their, because its not matching the images in your thesis, even if you add the missing \( \frac{\cdot}{\cdot} \) for the radial expansion. The images in your thesis consist of far more loops.
Figure 21: The runtime per loop of the density estimation. All configurations were tested 30 times, the displayed line is the mean of all tests.

Figure 22: The runtime per loop of the density estimation. All configurations were tested 30 times, the displayed line is the mean of all tests.
Figure 23: The runtime per loop of the density estimation. All configurations were tested 30 times, the displayed line is the mean of all tests.

**Conclusion**  Figure 24 shows the result of a speed analysis of the Matlab code (tested for high values of $p$ and $b$).

```matlab
1 tmp = repmat(prod(G(ID~=d,:),1),[size(Phi{d},1) 1]); %around 7% of the execution time
2 buc = iC{d} * mean(Phi{d} .* tmp,2); %around 15% of the execution time
```

Figure 24: The most expensive statement for the density estimation algorithm. Around 20% of the execution time are used for this calculation.

A part of these two code lines is the calculation of the product along the columns of matrix, but excluding one row. Matlab provides a fast implementation for this case, Eigen doesn’t. Therefore a simple loop algorithm was created for this part. It is probably slower then the corresponding Matlab Code and might be the reason for the bad scaling of the C++ algorithm with $p$. Similar functions were used in the regression implementation as well.

All in all scales the runtime of the density estimation algorithm strong with the number of points and Fourier bases. The number of dimension is not an important scaling parameter.

The C++ implementation is only faster then Matlab for small inputs with very low number of points and Fourier bases, but can reduce the runtime in these cases by more then 90%.

### 3.3.2 Accuracy

The density estimation algorithm was implemented last and couldn’t be tested as much as the other two algorithms. To estimate the error between the results of both algorithms, both created LFFs were
evaluated at 1000 random points (created by Matlab rand over the range of \([-3.5, 3.5]\)) and the mean absolute difference of these estimations was saved for each test.

Figure 25 shows the difference while changing the number of input points and the number of Fourier Bases. Most tests results in a very small difference, but each configurations had tests with a mean difference of over $10^{-8}$.

![Figure 25: The mean absolute difference between Matlab and C++ for the density estimation. All configurations were tested 30 times, the displayed line is the mean of all tests.](image)

Figure 25: The mean absolute difference between Matlab and C++ for the density estimation. All configurations were tested 30 times, the displayed line is the mean of all tests.

Figure 26 shows the difference while testing the number of dimensions. Surprisingly declines the difference between Matlab and C++ with an increase in the number of input dimension.

![Figure 26: The runtime per loop of the density estimation. All configurations were tested 30 times, the displayed line is the mean of all tests.](image)

Figure 26: The runtime per loop of the density estimation. All configurations were tested 30 times, the displayed line is the mean of all tests.

The unexpected high difference between Matlab and C++ might indicate that the C++ implementation has still some small errors, or just be a product of the numerical differences between Matlab and C++ (like in the other two algorithms). Unfortunately there wasn’t enough time to tests this implementation further.
4 Conclusion

The conclusion is separated for compression, regression and density estimation:

4.1 Compression

The speed of the first part of the compression algorithm, the calculation of the function norm, depends mostly on the number of factored functions in the input function. For a high number of factored functions is the provided Matlab implementation without the covariance matrix the fastest implementation, even faster than the GPU implementations. For a small number of factored functions provide the GPU implementations an enormous increase in computation speed.

The memory consumption for the calculation of the function norm scales quadratic with the number of factored functions, thus limiting the possible size of the input function. This is especially a problem for working with the GPU. The provided AdaptMemory algorithm circumvents this problem while providing nearly the same calculation speed as the provided Matlab implementation.

All in all performs the C++ implementation worse than the provide Matlab implementation for the calculation of the function norm.

The speed of the second part, the loop iterations, depends mostly on the number of factored functions and Fourier bases. The C++ implementation is faster than Matlab and scales equally for the number of factored functions. Matlab scales for the number of Fourier bases better than C++ and can therefore become faster if more than \( \approx 10000 \) Fourier bases are used.

The GPU implementation is dominated by the cost of the transfer between CPU and GPU for smaller input sized, but outperforms Matlab and C++ for inputs with more than 6000 Fourier bases or 2000 factored functions.

4.2 Regression

The C++ implementation of the regression algorithm is faster than the reference MATLAB version for small inputs in the number of data points, and we also achieved a speedup of up to 3 in the number of dimensions, which constantly grows with the number of dimensions. So the C++ implementation is recommended for high dimensional input spaces with relatively smaller number of samples available.

4.3 Density Estimation

The implementation of the density estimation algorithm is only for small inputs faster than Matlab.

The mean absolute difference between both algorithm is quite high, but the results were visual equal for all tests.
References


A Appendix

Figure 27: Execution time for solving systems of linear equations. A was generated randomly by the Matlab uniform number generator rand. Each matrix size was tested 50 times, the provided lines display the mean value of these tests. Eigen was compiled with MKL. The timing was analysed using the same test system as used for the compression algorithm. See Section 3.1.2 for details.

Figure 28: The mean absolute error while solving systems of linear equations. A was generated randomly by the Matlab uniform number generator rand. The error is defined as the mean of A * X - B. Each matrix size was tested 50 times. The error was analysed using the same test system as used for the compression algorithm. See Section 3.1.2 for details.
function fnorms = computeNormLowRamAdapt(fA, fB)
dim = size(fB, 1);
m = size(fA, 1);
AC = ones(1, m);

%choose k
availableMem = 4 * 1024 * 1024 * 1024; %4 GB
k = floor(availableMem / m / 8 / 2);
k = min(k, m);

if k == 0
    fnorms = nan;
else
    i = 1;
    for j = 1:k:m
        ub = min(i + k, m);
        fC = ones(ub - j + 1);
        for d = 1:dim
            fC = fC .* (fB{d}' * fB{d}(j:ub));
        end
        AC(1, j:ub) = fA' * fC;
        i = i + 1;
    end
    fnorms = AC * fA;
end

Figure 29: The Matlab code for the function norm calculation using the adapt iteration.

Figure 30: The calculation time of the function norm for the compression algorithm over the number of factored functions m.