Parallel Approach to Derivative-Free Optimization

Implementing the DONE Algorithm on a GPU

J.H.T. Munnix

Master of Science Thesis

TU Delft Delft University of Technology

Delft Center for Systems and Control

Parallel Approach to Derivative-Free Optimization

Implementing the DONE Algorithm on a GPU

Master of Science Thesis

For the degree of Master of Science in Systems and Control at Delft University of Technology

J.H.T. Munnix

September 26, 2016

Faculty of Mechanical, Maritime and Materials Engineering (3mE) · Delft University of Technology

Abstract

Researchers at Delft University of Technology have recently developed an algorithm for optimizing noisy, expensive and possibly nonconvex objective functions for which no derivatives are available. The data-based online nonlinear extremum-seeker (DONE) was originally developed for sensorless wavefront aberration correction in optical coherence tomography (OCT) and optical beam forming network (OBFN) tuning. In order to make the DONE algorithm suitable for large-scale problems, a parallel implementation using a graphics processing unit (GPU) is considered. This master thesis aims to develop such a parallel implementation which performs faster than the existing sequential implementation without much change in obtained accuracy. Since OBFN tuning is a problem that may involve a large amount of parameters, an OBFN simulation is to be used to compare the parallel implementation to the sequential implementation. The key of the DONE algorithm is solving a regularized linear least-squares problem in order to construct a smooth and low-cost surrogate function which does provide derivatives and can be optimized fairly easily. This master thesis first discusses the basics of parallel computing, after which several linear least-squares methods and several numerical optimization methods are investigated. These methods are compared and the most suitable methods for parallel computing are implemented and tested for increasing dimensions. The final parallel DONE implementation combines the recursive least-squares (RLS) method with the Broyden-Fletcher-Goldfarb-Shanno (BFGS) method and optimizes the large-scale OBFN simulation almost twice as fast as the sequential DONE implementation, without much change in obtained accuracy.

Table of Contents

List of Figures

List of Tables

Acknowledgements

Before you lies the result of ten months of exploration through the large territories of linear algebra and scientific computing. It all started on December 7, 2015, which was the official first day of my graduation project for the master Systems & Control. One might say that after six years of scientific education there should not be much more to discover, however this is not the case. I have the feeling that during these ten months I have discovered so much more, things I never even expected they existed.

And now my project is finished and it is time to start focussing on new challenges. I hope to be able to keep on learning new things, things for which these past six years have been a solid base of knowledge. I would like to thank my daily supervisors MSc. H.R.G.W. Verstraete and MSc. L. Bliek for their guidance and I also would like to thank my thesis advisor Prof.Dr.Ir. M. Verhaegen for allowing me to work on this project in the first place. Now the only thing left to do is present my work, which I hereby do, hoping that you will enjoy reading it.

Delft University of Technology J.H.T. Munnix
September 26, 2016

If you know how to measure success and, more importantly, its derivative, you are already halfway to achieving your goals. All that is left to do, is choosing your favourite descent direction.

Chapter 1

Introduction

The DONE algorithm

Researchers at Delft University of Technology have recently developed an algorithm for optimizing objective functions that can be measured in practice but are difficult to model [1, 2, 3]. The objective function in question may be expensive, noisy and nonconvex. Moreover no derivatives are available, which classifies the algorithm as a derivative-free optimization algorithm [4]. The algorithm is referred to as data-based online nonlinear extremum-seeker (DONE) and aims to converge to the global optimum on a predefined domain, within a minimal amount of function evaluations. Possible applications for the algorithm are sensorless wavefront aberration correction in optical coherence tomography (OCT) [2] and tuning of an optical beam forming network (OBFN) [3].

The derivative-free character of the DONE algorithm is useful for objective functions that represent the performance of a simulation or an experimental application. In these situations an analytical expression for the derivative is not available and approximating derivatives with finite difference computations is either very expensive, in case the measurements are expensive, or it does not provide reliable results, in case the measurements are noisy. The online character of the DONE algorithm allows immediate achievement of results, which improve in accuracy as the amount of measurements increases [1].

The key of the DONE algorithm is constructing a surrogate function using a random Fourier expansion (RFE) [5]. This surrogate function does provide derivatives and can be optimized fairly easily using standard numerical optimization methods [6], after which the found optimum is used to acquire a new measurement. In order to deal with possible nonconvexity, random exploration steps are added to evaluation points.

Since practical optimization problems may consist of a large amount of parameters, the computations required to perform a single cycle of the DONE algorithm may become very computationally expensive. Moreover, when the amount of parameters increases, inevitably the amount of required cycles increases as well. Hence performing the DONE algorithm for large-scale problems may require a lot of time. A parallel implementation of the DONE algorithm may yield an improvement in this case.

Parallel computing has recently gained a large amount of interest since sequential computing does not seem to have much more improvement possibilities with respect to computation speed whereas parallel computing does [7, 8]. A commonly used device for parallel computing is a graphics processing unit (GPU). Originally GPUs were designed for rendering high-resolution graphics at high frame rates, however nowadays they are becoming more and more used in scientific computing as well [9]. Where a central processing unit (CPU) performs computations sequentially, a GPU performs them simultaneously. For large-scale problems, using a GPU may result in a large speed-up with respect to only using a CPU. A commonly used platform for parallel computing is the compute unified device architecture (CUDA) platform, which is developed by NVIDIA® [10, 11].

Research questions

The goal of this thesis is to come up with a parallel implementation of the DONE algorithm on a GPU using CUDA [9, 10, 11]. Moreover the parallel implementation must be proven to require less run time for a practical large-scale problem than the existing sequential implementation, while maintaining similar accuracy of results. In order to achieve this goal, a main research question and several sub-questions are to be answered:

> What are the possibilities of parallelization of the DONE algorithm by means of implementing it on a GPU using CUDA, with the focus on achieving a maximum speed-up factor for large-scale problems, with respect to the existing sequential implementation, while maintaining correct results?

1. In which scenario is the DONE algorithm used and in which scenario can a parallel DONE implementation be useful?
2. What are the core elements of the DONE algorithm?
3. How can the DONE algorithm be classified and which other algorithms belong to the same classification?
4. What are the key concepts of parallel computing on a GPU using CUDA?
5. What are the criteria that determine whether or not a numerical method is suitable for parallel computing?
6. What are the available numerical methods that can be used for the core elements and how do they compare with respect to the established criteria?
7. What are the criteria that are important for comparison between sequential and parallel implementations of the selected methods?
8. Which of the selected methods perform best according to the established criteria?
9. How does the acquired parallel implementation of the DONE algorithm compare to the existing sequential implementation?

Report structure

The sub-questions are to be used as a guideline for the project. The scenario for the sequential and parallel DONE algorithm is discussed in Chapter 2. Chapter 3 discusses the core elements of the DONE algorithm and its classification among other methodology. The key concepts of parallel computing and the parallel computing criteria are discussed in Chapter 4. In Chapters 5 and 6 the available numericals methods for the core elements are discussed and they are judged based on the established criteria. Subsequently the implementation of the selected numerical methods is discussed in Chapter 7. Once the implementation is completed successfully, the implementations are to be tested using several experiments and judged based on several experiment criteria, which are discussed in Chapter 8. The results of these experiments are discussed in Chapter 9. Finally, Chapters 10 and 11 summarize the answers to the sub-questions and answer the main research question. Also an evaluation and recommendations for future work are presented.

Chapter 2

Scenario Description

This chapter describes the scenario in which the DONE algorithm can be used and extends this scenario to include situations in which a parallel implementation of the DONE algorithm may be useful. DONE stands for data-based online nonlinear extremum-seeker and is developed for solving the optimization problem

$$\min_{\boldsymbol{x}} f(\boldsymbol{x}) \quad \text{s.t.} \quad \boldsymbol{x} \in \mathcal{X}, \tag{2-1}$$

where $f : \mathbb{R}^n \mapsto \mathbb{R}$ and $\mathcal{X} \subset \mathbb{R}^n$. This is the basic description, however the DONE algorithm is developed for the specific situation where no derivatives are available and function evaluations are noisy and expensive. Here noisy means that there are slight variations between multiple measurements at the same point and expensive means that measurements are money-consuming, time-consuming or both. The DONE algorithm also has some tools to deal with nonconvexity, however this is not one of its main focus points [1].

The DONE algorithm aims to solve the optimization problem (2-1) within as few function evaluations as possible. It does this by applying function fitting in combination with optimization. A characteristic of the DONE algorithm is that is has a fixed computation time per cycle. Special structures or sparsity are not exploited. The DONE algorithm can be used in any situation that meets the description above, however it was originally developed for sensorless wavefront aberration correction in optical coherence tomography (OCT) [2] and optical beam forming network (OBFN) tuning [3].

Since OBFN tuning is an application that may involve dimensions of the order $n > 100$ and function evaluations are expensive, running the DONE algorithm may take a lot of time for this application. This is due to the fact that the computations that are required for the DONE algorithm depend strongly on the dimension n and the fact that increasing the dimension of an optimization problem also increases the required amount of optimization cycles. A detailed explanation of how the dimension n is related to the required operations for the DONE algorithm is given in Chapter 3.

Recently, parallel computing has been gaining more and more interest, since it has very good possibilities of speeding up existing technology and because sequential computing appears to have reached its computation speed limit [7, 8, 9]. Especially for large-scale problems parallel computing may yield a large speed-up, therefore a parallel DONE implementation may be able to perform OBFN tuning faster than the current sequential DONE implementation. Figure 2-1 shows a summary of the scenario in which the DONE algorithm can be used and the scenario in which a parallel implementation of the DONE algorithm may be useful.

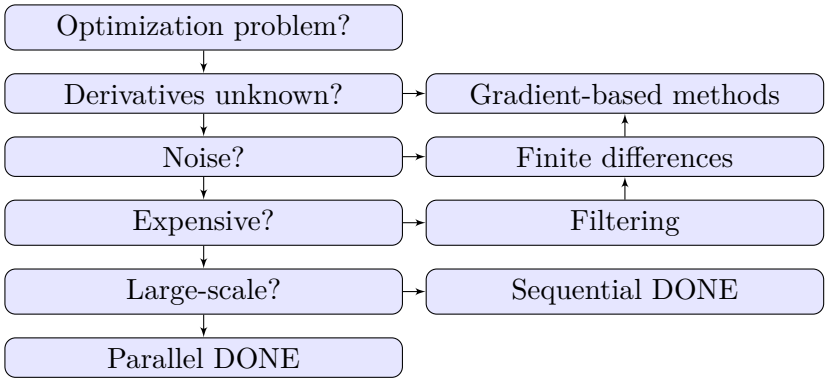


Figure 2-1: Scenario for using the DONE algorithm [1]. If the answer is 'yes' follow the arrows downward, if the answer is 'no' switch from the left column to the right column and follow the arrows upward.

Naturally, DONE is not the only method that exists for solving a derivative-free optimization problem. Derivative-free optimization is a difficult task, to which already a lot of research has been devoted [4]. The DONE algorithm can be classified as a model-based derivative-free optimization algorithm. The model that is used is however not based on dynamical properties, but solely on available data, which explains the contrasting term "data-based" in the acronym DONE.

Chapter 3

The DONE Algorithm

This chapter discusses the core elements of the DONE algorithm. DONE is developed for optimizing any objective function $f : \mathbb{R}^n \mapsto \mathbb{R}$ that is noisy, expensive and possibly nonconvex, moreover neither analytical derivatives, nor numerical derivatives are available [1]. The problem at hand is denoted by (2-1) where the feasible region $\mathcal{X} \subset \mathbb{R}^n$ is defined by a lower bound $\boldsymbol{x}_{lb} \in \mathbb{R}^n$ and an upper bound $\boldsymbol{x}_{ub} \in \mathbb{R}^n$, ensuring that

$$\boldsymbol{x}_{lb} \leq \boldsymbol{x} \leq \boldsymbol{x}_{ub} \quad \forall \quad \boldsymbol{x} \in \mathcal{X}, \tag{3-1}$$

where the $\leq$ operator is an element-wise operator.

3-1 Flowchart

Figure 3-1 shows a flowchart representing the DONE algorithm. Each cycle a surrogate function $\hat{f}_i : \mathbb{R}^n \mapsto \mathbb{R}$ is constructed based on available measurements, where $i \in \mathbb{Z}$ counts the amount of cycles and measurements. Consequently, per cycle a single measurement is obtained. Contrary to the objective function, the surrogate function is smooth, low-cost and does provide derivatives, hence it can be optimized in a rather simple fashion. Subsequently a new measurement is acquired and the procedure is repeated, ultimately converging to the optimum. The surrogate function $\hat{f}_i$ may however be nonconvex within $\mathcal{X}$, hence multiple local optima may exist. To prevent getting stuck in a local optimum, DONE adds small perturbations to the evaluation points.

3-2 RFE Fitting

The DONE algorithm constructs a surrogate function $\hat{f}_i$ using a random Fourier expansion (RFE) [5], which can be interpreted as a single-layered neural network [12, 13] with randomly selected weights and biases and a trigonometric activation function.

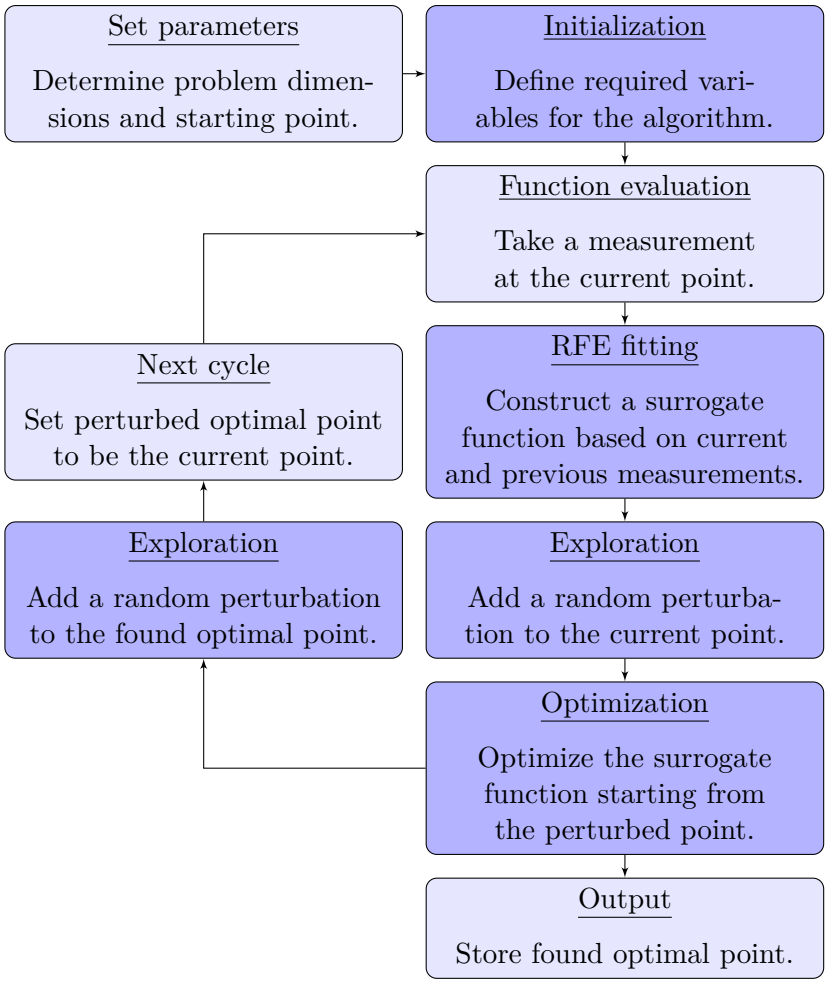


Figure 3-1: This flowchart represents the DONE algorithm. The dark shaded steps are the core elements and the steps which are susceptible for parallel computing.

In this case, a neural network with $n \in \mathbb{Z}$ inputs and a single output is considered, making it suitable as a surrogate function for the objective function f. The single-layered neural network $\psi : \mathbb{R}^n \mapsto \mathbb{R}$ with $m \in \mathbb{Z}$ neurons and activation function $\varphi : \mathbb{R} \mapsto \mathbb{R}$ is denoted by

$$\psi(\boldsymbol{x}) = \sum_{j=1}^{m} w_j \varphi(\boldsymbol{\omega}_j^T \boldsymbol{x} + b_j) = \boldsymbol{w}^T \varphi(\boldsymbol{\Omega}^T \boldsymbol{x} + \boldsymbol{b}), \tag{3-2}$$

where the respective input weights, biases and output weights are represented by

$$\boldsymbol{\Omega} = \begin{bmatrix} \boldsymbol{\omega}_1 & \dots & \boldsymbol{\omega}_m \end{bmatrix}, \quad \boldsymbol{b} = \begin{bmatrix} b_1 & \dots & b_m \end{bmatrix}^T, \quad \boldsymbol{w} = \begin{bmatrix} w_1 & \dots & w_m \end{bmatrix}^T.$$

Every time a new set of input and output measurements $\{\boldsymbol{x}_i \in \mathcal{X}, y_i \in \mathbb{R}\}$ becomes available, the weights and biases can be determined by the nonlinear least-squares problem

$$\underset{\boldsymbol{\Omega}, \boldsymbol{b}, \boldsymbol{w}_i}{\operatorname{argmin}} \; ||\boldsymbol{y}_i - \boldsymbol{\psi}_i||_2^2, \tag{3-3}$$

where all measurements are gathered into

$$\boldsymbol{y}_i = \begin{bmatrix} y_1 & \dots & y_i \end{bmatrix}^T, \quad \boldsymbol{\psi}_i = \begin{bmatrix} \psi(\boldsymbol{x}_1) & \dots & \psi(\boldsymbol{x}_i) \end{bmatrix}^T.$$

For standard neural networks, this problem is often solved using nonlinear optimization methods [13]. For the RFE however, the entries of $\boldsymbol{\Omega} \in \mathbb{R}^{n \times m}$ and $\boldsymbol{b} \in \mathbb{R}^m$ are not optimized but drawn from a certain probability distribution. By defining

$$\boldsymbol{a}_i = \begin{bmatrix} \varphi(\boldsymbol{\omega}_1^T \boldsymbol{x}_i + b_1) \\ \vdots \\ \varphi(\boldsymbol{\omega}_m^T \boldsymbol{x}_i + b_m) \end{bmatrix} = \varphi(\boldsymbol{\Omega}^T \boldsymbol{x}_i + \boldsymbol{b}), \tag{3-4}$$

$$\boldsymbol{A}_i = \begin{bmatrix} \varphi(\boldsymbol{\omega}_1^T \boldsymbol{x}_1 + b_1) & \dots & \varphi(\boldsymbol{\omega}_m^T \boldsymbol{x}_1 + b_m) \\ \vdots & \ddots & \vdots \\ \varphi(\boldsymbol{\omega}_1^T \boldsymbol{x}_i + b_1) & \dots & \varphi(\boldsymbol{\omega}_m^T \boldsymbol{x}_i + b_m) \end{bmatrix} = \begin{bmatrix} \boldsymbol{a}_1^T \\ \vdots \\ \boldsymbol{a}_i^T \end{bmatrix}, \tag{3-5}$$

where $\boldsymbol{A}_i \in \mathbb{R}^{i \times m}$ now only contains known data, the optimal output weight vector $\boldsymbol{w}_i \in \mathbb{R}^m$ can be determined by the regularized linear least-squares problem

$$\boldsymbol{w}_i = \underset{\boldsymbol{w}}{\operatorname{argmin}}\, J_i(\boldsymbol{w}), \tag{3-6}$$

$$J_i(\boldsymbol{w}) = ||\boldsymbol{y}_i - \boldsymbol{A}_i \boldsymbol{w}||_2^2 + \lambda \, ||\boldsymbol{w}||_2^2, \tag{3-7}$$

where $J_i : \mathbb{R}^m \mapsto \mathbb{R}$ is the least-squares cost function and $\lambda \in \mathbb{R}$ is the regularization parameter [14]. This procedure yields a surprisingly good surrogate function if the activation function is chosen to be an infinitely differentiable function [12]. Moreover this procedure is much faster and less complex than fully optimizing the weights and biases using the nonlinear least-squares problem (3-3) [15]. By choosing the activation function to be

$$\varphi(\boldsymbol{\omega}_j^T \boldsymbol{x}_i + b_j) = \cos(\boldsymbol{\omega}_j^T \boldsymbol{x}_i + b_j), \tag{3-8}$$

which is infinitely differentiable, and by drawing $\boldsymbol{\omega}_j \sim \mathcal{N}(\boldsymbol{0}_n, \sigma_{\boldsymbol{\omega}}^2 \boldsymbol{I}_n)$ and $b_j \sim \mathcal{U}(0, 2\pi)$, where $\mathcal{N}$ indicates a normal probability distribution with a mean and a covariance respectively and $\mathcal{U}$ indicates a uniform probability distribution with a lower and an upper bound respectively, the value of $\sigma_{\boldsymbol{\omega}}$ can be chosen such that the output of the surrogate function only contains frequencies from a desired frequency range [1]. Now the surrogate function is denoted by

$$\hat{f}_i(\boldsymbol{x}) = \boldsymbol{w}_i^T \cos(\boldsymbol{\Omega}^T \boldsymbol{x} + \boldsymbol{b}). \tag{3-9}$$

Contrary to the objective function f, the surrogate function $\hat{f}_i$ is not expensive, not noisy and does provide derivatives (3-12). It may however still be nonconvex, which is important to keep in mind.

From here on, in order to fit into the RFE framework, $\boldsymbol{\Omega}$ will be referred to as the frequency matrix, $\boldsymbol{b}$ will be referred to as the phase vector, $\boldsymbol{w}$ will be referred to as the weight vector and m will be referred to as the amount of Fourier features.

3-3 Optimization

Since the DONE algorithm optimizes the surrogate function $\hat{f}_i$ instead of the objective function f, the optimization problem denoted by

$$\boldsymbol{x}_{i+1} = \underset{\boldsymbol{x}}{\operatorname{argmin}}\, \hat{f}_i(\boldsymbol{x}) \quad \text{s.t.} \quad \boldsymbol{x} \in \mathcal{X}, \tag{3-10}$$

$$\hat{f}_i(\boldsymbol{x}) = \boldsymbol{w}_i^T \cos(\boldsymbol{\Omega}^T \boldsymbol{x} + \boldsymbol{b}) \tag{3-11}$$

is the problem that has to be solved every i^{th} cycle. The surrogate function $\hat{f}_i$ does provide derivatives, for instance the gradient and Hessian are denoted by

$$\nabla_{\boldsymbol{x}} \hat{f}_i(\boldsymbol{x}) = -\boldsymbol{\Omega}\, \operatorname{diag}(\boldsymbol{w}_i) \sin(\boldsymbol{\Omega}^T \boldsymbol{x} + \boldsymbol{b}), \tag{3-12}$$

$$\nabla^2_{\boldsymbol{x}} \hat{f}_i(\boldsymbol{x}) = -\boldsymbol{\Omega}\, \operatorname{diag}(\boldsymbol{w}_i) \operatorname{diag}(\cos(\boldsymbol{\Omega}^T \boldsymbol{x} + \boldsymbol{b}))\boldsymbol{\Omega}^T. \tag{3-13}$$

Here the diag() operator transforms a vector into a diagonal matrix, which ensures that the gradient and Hessian have the proper dimensions.

3-4 Exploration

Due to the possible nonconvexity of f and $\hat{f}_i$, multiple local optima may exist within $\mathcal{X}$. To prevent getting stuck in a local optimum, before starting the optimization step, a random vector $\boldsymbol{\zeta}_i \sim \mathcal{N}(\boldsymbol{0}_n, \sigma^2_{\boldsymbol{\zeta}} \boldsymbol{I}_n)$ is drawn and the current evaluation point is perturbed by

$$\boldsymbol{x}_i = \min(\max(\boldsymbol{x}_i + \boldsymbol{\zeta}_i, \boldsymbol{x}_{lb}), \boldsymbol{x}_{ub}), \tag{3-14}$$

ensuring that $\boldsymbol{x}_i \in \mathcal{X}$ and possibly jumping out of a local optimum of $\hat{f}_i$. Moreover in order to determine the next evaluation point, the found optimal point $\boldsymbol{x}_{i+1} \in \mathcal{X}$ is perturbed by a randomly drawn vector $\boldsymbol{\xi}_i \sim (\boldsymbol{0}_n, \sigma^2_{\boldsymbol{\xi}} \boldsymbol{I}_n)$, which leads to

$$\boldsymbol{x}_{i+1} = \min(\max(\boldsymbol{x}_{i+1} + \boldsymbol{\xi}_i, \boldsymbol{x}_{lb}), \boldsymbol{x}_{ub}), \tag{3-15}$$

ensuring that $\boldsymbol{x}_{i+1} \in \mathcal{X}$ and possibly jumping out of a local optimum of f. By choosing large values of $\sigma_{\boldsymbol{\zeta}}$ and $\sigma_{\boldsymbol{\xi}}$ a rough location of the global optimum may be found and by choosing small values of $\sigma_{\boldsymbol{\zeta}}$ and $\sigma_{\boldsymbol{\xi}}$ a detailed location of a possibly local optimum may be found. In practice the values are often chosen as $\sigma_{\boldsymbol{\zeta}} = \sigma_{\boldsymbol{\xi}}$ [1].

3-5 Comparable Methodology

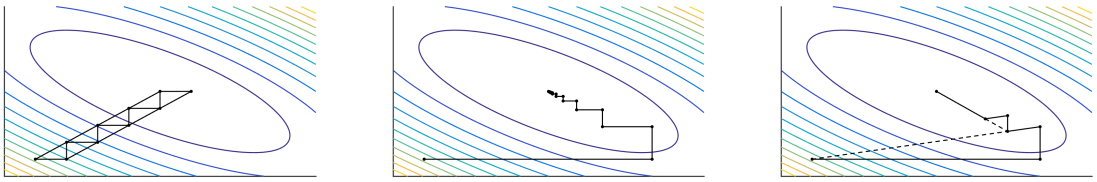
As discussed in Chapter 2 the DONE algorithm is not the only method for performing derivative-free optimization. Derivative-free optimization methods can be classified as either data-based or model-based. Data-based methods, also referred to as direct methods, determine search directions by computing values of the function f directly, whereas model-based methods construct a surrogate function for f to guide the search process [4]. Examples of data-based methods are the Nelder-Mead simplex (NMS) method [16], the coordinate search (CS) method [4] and Powell's conjugate directions (PCD) method [17]. Examples of model-based methods are NEWUOA [18] (acronym not specified in literature), Bayesian optimization [19] and DONE. As discussed in Chapter 2 this classification is despite the fact that DONE stands for data-based online nonlinear extremum-seeker.

Data-based methods

The NMS method uses a set of $n+1$ points from which a so-called simplex can be constructed. For each point a measurement is taken and this information is used to replace one of the points with a new point with a lower function value. This procedure is repeated until a local optimum is reached [4, 6, 16]. Figure 3-2a illustrates the NMS method

The CS method repeatedly optimizes a function along each of its coordinate directions using an exact or inexact line search method [6]. The coordinate search method is very similar to the gradient descent (GD) method, discussed in Section 6-3-1, except that no gradients need to be computed [4, 6]. Figure 3-2b illustrates the CS method.

Where the coordinate search method is similar to the GD method, the PCD method is similar to the conjugate gradient (CG) method, discussed in Section 6-3-1 as well. The PCD method is able to construct conjugate line search directions from previously used directions, however without requiring the computation of the gradient [6, 17]. Figure 3-2c illustrates the PCD method.



(a) NMS method **(b)** CS method **(c)** PCD method

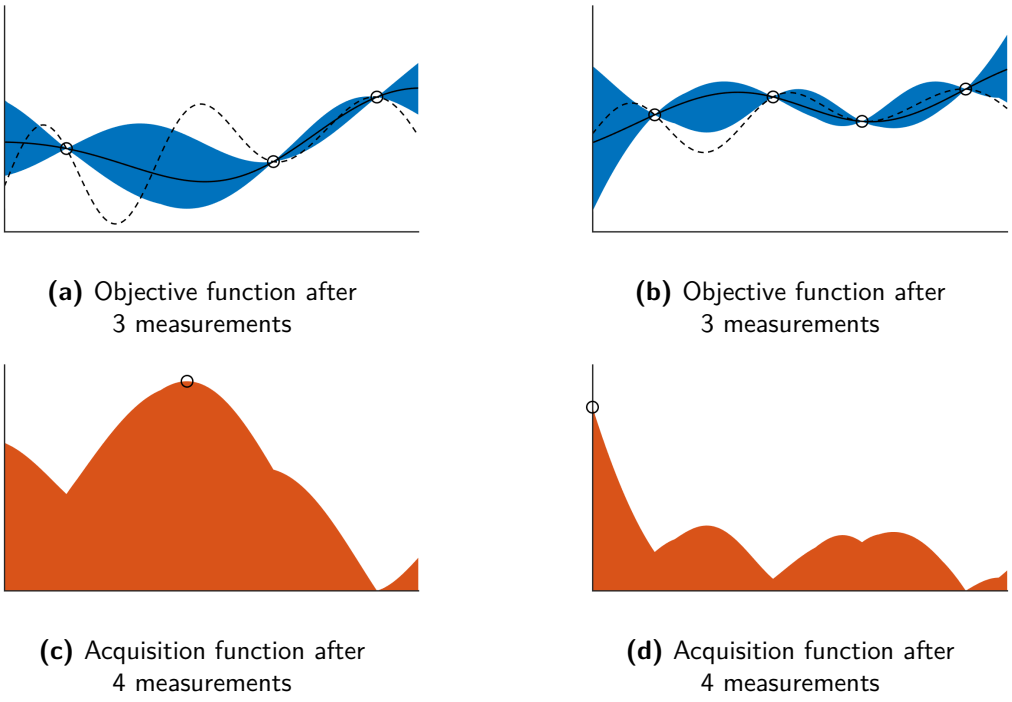
Figure 3-2: Data-based derivative-free optimization of a two-dimensional quadratic function using the Nelder-Mead simplex (NMS) method, the coordinate search (CS) method and Powell's conjugate directions (PCD) method.

Model-based methods

NEWUOA approaches the problem in a similar fashion to the DONE algorithm. However, instead of constructing a surrogate function using a RFE, a quadratic surrogate function

is constructed [18]. For this quadratic surrogate function an analytical expression of the optimum is available. Moreover some clever tricks are used to allow function fitting with only a few measurements and to forget measurements that are no longer relevant.

Bayesian optimization is an optimization method which uses Bayes' theorem [19]. Using a set of available measurements, a Gaussian process approximation is obtained. The mean of this Gaussian process can be considered as a surrogate function. It is however not the surrogate function that is optimized in Bayesian optimization. Both the mean and the covariance of the Gaussian process are combined in what is referred to as the acquisition function. By optimizing this acquisition function, a trade-off is made between exploration and exploitation. Finding points where the surrogate function has low values is referred to as exploitation, whereas finding points where the uncertainty is high is referred to as exploration. Optimizing the acquisition function instead of the surrogate function has the property that it aims to minimize the number of measurements. Moreover it is likely to perform well for objective functions with multiple local minima [19]. Figure 3-3 illustrates Bayesian optimization.



(a) Objective function after 3 measurements

(b) Objective function after 3 measurements

(c) Acquisition function after 4 measurements

(d) Acquisition function after 4 measurements

Figure 3-3: These are two steps in a one-dimensional Bayesian optimization process. The dashed line represents the objective function, the solid line represents the mean of the Gaussian process and the shaded areas represent the covariance of the Gaussian process and the acquisition function.

Comparison

An advantage of NEWUOA over DONE is that NEWUOA provides an analytical expression for the optimum of the surrogate function, hence it requires no iterative optimization method. However, this yields a disadvantage as well since the surrogate function based on a RFE used by DONE is able to globally approximate any continuous function [20] whereas the convex quadratic surrogate function used by NEWUOA can only do this locally [18]. A difference

between NEWUOA and DONE is that NEWUOA dismisses measurements when they are no longer relevant dan DONE does not. However when necessary this can be easily incorporated into the DONE algorithm by using a so-called forgetting factor in the RFE fitting step [14].

A key feature of Bayesian optimization is that it optimizes an acquisition function instead of a surrogate function. This allows for exploitation and exploration, which are very suitable concepts for finding the global optimum of a nonconvex function. DONE applies two exploration steps to achieve the same result, however the Bayesian approach may be applicable to DONE as well. The advantage of using a Gaussian process instead of a RFE, is that the Gaussian process is very useful for establishing the acquisition function. An advantage of DONE over Bayesian optimization is that for DONE the computation time is independent of the amount of measurements [1] whereas for Bayesian optimization this not the case [19].

Chapter 4

Parallel Computing

This chapter discusses the concept of parallel computing and in which cases it may or may not be useful. Although multiple possibilities exist for parallel computing [7], the focus lies on graphics processing unit (GPU) programming using the platform compute unified device architecture (CUDA) [10, 11]. Furthermore a set of criteria is established which can be used to determine whether or not a numerical method is suitable for parallel computing.

4-1 Basic Concepts of Computing

When developing a numerical algorithm, one may want to make it as fast as possible, in order to save time, thus save money. Hence useful pieces of information are how the time required to run the algorithm is influenced by computation, communication and complexity.

Computation speed

Central processing unit (CPU) speed is measured by its clock frequency in clock cycles per second (or Hertz). The amount of computations that can be performed during each clock cycle may vary per type of CPU and the clock frequency may vary per type of CPU as well. In order to express the computation speed in terms that are more related to numerical methods, often the term floating-point operations per second (FLOPS) is used. The floating-point format is used to store real numbers. This format can either be single-precision or double-precision and, as the names suggest, double-precision floating-point numbers require twice as much storage space as single-precision floating-point numbers [21].

The basic floating-point operations are addition, subtraction and multiplication. These operations typically require one clock cycle. Division is an operation which often requires more clock cycles than the basic floating-point operations. Therefore, when performing many divisions by the same number, it may be wise to compute the inverse of this number once and use this value for multiplication [9].

Communication speed

Often communications are more of a limiting factor than computations. The communication time $t_{send} \in \mathbb{R}$ in seconds of $u \in \mathbb{Z}$ bits between two memory spaces is denoted by

$$t_{send} = \alpha + \beta u, \tag{4-1}$$

where $\alpha \in \mathbb{R}$ is the latency in seconds and $\beta \in \mathbb{R}$ is the bandwidth in seconds per bit [9]. The latency is the amount of time required to set up the connection and the bandwidth is the amount of time required to transfer a single bit. The communication speed $v_{send} \in \mathbb{R}$ in bits per second is denoted by

$$v_{send} = \frac{u}{t_{send}} = \frac{u}{\alpha + \beta u} \leq \frac{1}{\beta}, \tag{4-2}$$

where the upper limit is reached for the ideal case where $\alpha = 0$.

Computational complexity

The computational complexity of an algorithm is measured by the amount of operations it performs. Although this is not limited to floating-point operations, the floating-point operations do provide a good measure for numerical algorithms. Computational complexity is always measured as a function of the limiting dimensions. For the DONE algorithm these dimensions are the input dimension n and the amount of Fourier features m. By assuming the problem is large-scale, the computational complexity gives a good measure for how an algorithm performs for $n \to \infty$ and $m \to \infty$.

4-2 Basic Concepts of Parallelism

Flynn's categories

Computers can be categorized in four categories, known as Flynn's categories [22]:

- SISD: single instruction stream, single data stream
- SIMD: single instruction stream, multiple data stream
- MISD: multiple instruction stream, single data stream
- MIMD: multiple instruction stream, multiple data stream

Classical sequential computers can be categorized under the SISD category. Since GPUs contain many processors, which all perform the same operation on different chunks of data, they are an example of SIMD computing. MISD computers are rarely built and MIMD computers are the most general form of computers, however there are few classes of problems that require the MIMD category [9, 22].

Degree of parallelism

The degree of parallelism D of a numerical method is defined as the number of operations divided by the number of required parallel stages [23]. For instance the addition of two vectors $\boldsymbol{a} \in \mathbb{R}^u$ and $\boldsymbol{b} \in \mathbb{R}^u$ to produce a third vector $\boldsymbol{c} \in \mathbb{R}^u$ using

$$c_j = a_j + b_j \quad \forall \quad j \in \{1, \ldots, u\} \tag{4-3}$$

can be done using u parallel operations and only one stage is required, which results in

$$D = \frac{u}{1} = u.$$

On the contrary the addition of all entries of a vector $\boldsymbol{a} \in \mathbb{R}^u$ to produce a scalar $\gamma \in \mathbb{R}$ using

$$\gamma = \sum_{j=1}^{u} a_j \tag{4-4}$$

appears to be computable only in a sequential fashion, since the result of $a_1 + a_2$ must be available before $a_1 + a_2 + a_3$ can be computed. This results in

$$D = \frac{u}{u} = 1.$$

There are clever tricks to improve the degree of parallelism, even for situations like (4-4) [23], however no full parallelization can be obtained here. Since for many algorithms it is rather complicated to accurately compute the degree of parallelism, from here on it is considered to be a qualitative property instead of a quantitative property. The key element in evaluating the degree of parallelism is the dependency between successive steps of an algorithm.

Computation time

The computation time in seconds of a numerical method executed on $\rho \in \mathbb{Z}$ parallel processors is denoted by $t_\rho \in \mathbb{R}$. For $u \in \mathbb{Z}$ operations the computation times can be computed using

$$t_1 = u\tau, \tag{4-5}$$

$$t_\rho = \left(\frac{u}{\rho} + \nu\right)\tau, \tag{4-6}$$

where $\tau \in \mathbb{R}$ is the computation time in seconds required for a single operation and $\nu \in \mathbb{R}$ is an overhead factor which is due to for instance data transfers and for which typically holds that $\nu > 1$ [23]. Hence the relation between t_ρ and t_1 becomes

$$t_\rho = \left(\frac{1}{\rho} + \frac{\nu}{u}\right) t_1 \geq \frac{t_1}{\rho}, \tag{4-7}$$

where the lower bound is reached for the ideal case in which $\nu = 0$.

Run time

Since there may be confusion about what is indicated by communication time and computation time, from here on the term run time is employed, which indicates all required communications and computations for a program to run.

Speed-up factor

The speed-up factor $S_\rho \in \mathbb{R}$ of a numerical method executed on $\rho \in \mathbb{Z}$ parallel processors with respect to the same method executed on a single processor is denoted by

$$S_\rho = \frac{t_1}{t_\rho} = \frac{u}{\frac{u}{\rho} + \nu} = \frac{u\rho}{u + \rho\nu} \leq \rho, \tag{4-8}$$

where $n \in \mathbb{Z}$ is the amount of operations and $\nu \in \mathbb{R}$ is the overhead factor [9, 23]. The upper bound is reached for the ideal case in which $\nu = 0$. After defining $\eta \in [0, 1]$ as the fraction of the work which is executed in parallel, the speed-up factor becomes

$$S_\rho = \frac{t_1}{\eta t_\rho + (1-\eta)t_1} = \frac{1}{\frac{\eta}{\rho} + \frac{\eta\nu}{u} + 1 - \eta} \leq \frac{1}{1-\eta}, \tag{4-9}$$

where the upper bound is reached for the ideal case in which $\nu = 0$ and $\rho \to \infty$. This means that for instance for a parallel work fraction $\eta = 0.9$ the speed-up factor will never exceed $(1 - 0.9)^{-1} = 10$ [9].

Efficiency

The efficiency $E_\rho \in [0, 1]$ of a numerical method executed on $\rho \in \mathbb{Z}$ processors with respect to the same method executed on a single processor is denoted by

$$E_\rho = \frac{S_\rho}{\rho} = \frac{u}{u + \rho\nu} \leq 1, \tag{4-10}$$

where the upper bound is reached for the ideal case in which $\nu = 0$.

Granularity

Granularity is the ratio of the amount of computation with respect to the amount of communication. Fine-grained parallelism means that the required computations are divided into small groups and communication occurs frequently, whereas coarse-grained parallelism means that communication occurs infrequently and is separated by large chunks of computations. Since on any computer communication is generally slower than computation, coarse-grained parallelism yields a higher speed-up factor, by allowing the required amount of communication to hide behind the larger required amount of computation [9]. This is an important consideration to make when deciding which parts of a numerical method are to be executed in parallel.

Performance gain

The most important thing to keep in mind is that parallel computing is only beneficial for large-scale problems. As can be seen in Figure 4-1, performing a computation on a CPU is generally faster than performing the same computation on a GPU. So only when multiple computations are to be performed, parallel computing may yield a speed-up. However there is also communication time to be considered. Before a GPU can perform computations, the required data need to be available to the GPU and afterwards the results need to be sent back to the CPU. Due to these necessities parallel computing only yields a speed-up for large-scale problems. As discussed, fine-grained parallelism requires communication before and after every computation and leads therefore to a smaller speed-up than coarse-grained parallelism, which only requires communications before and after a series of computations.

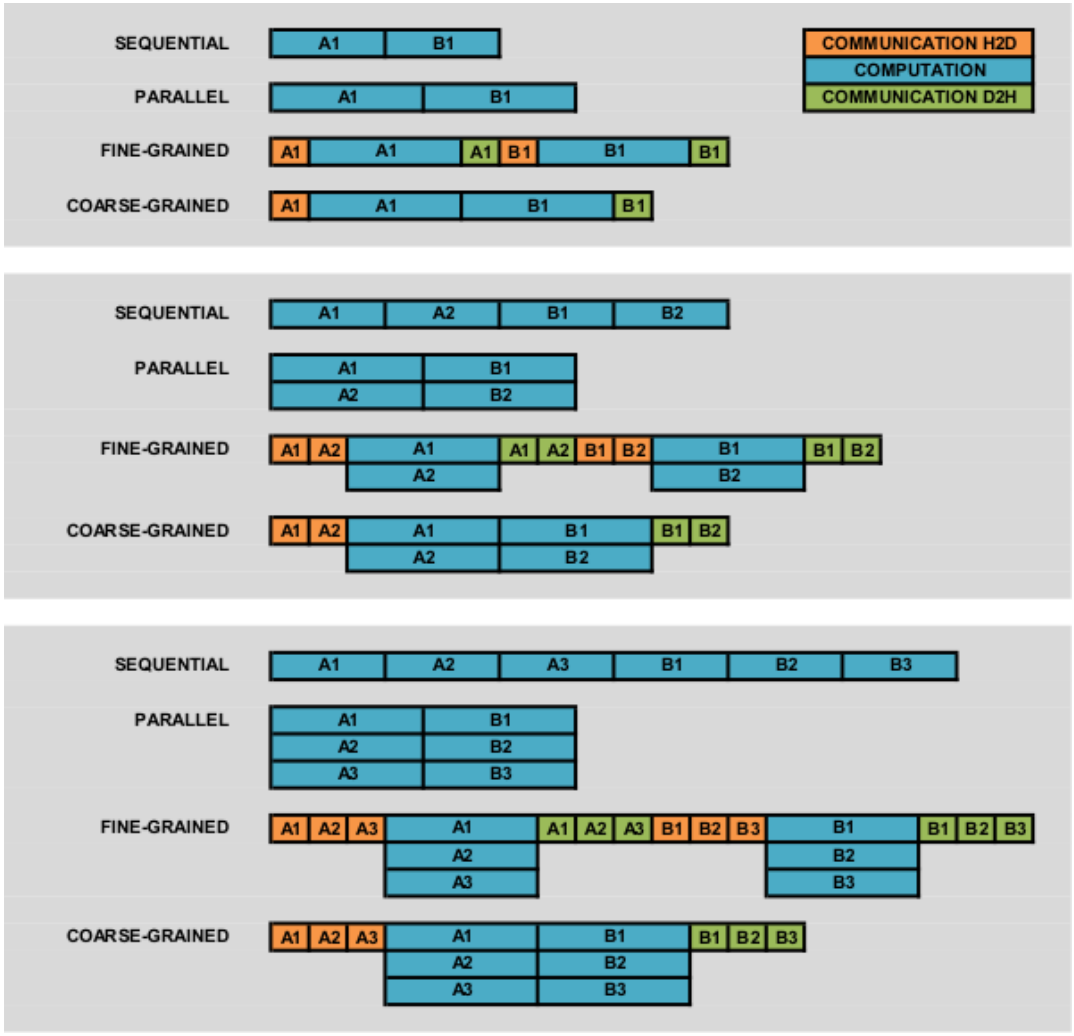
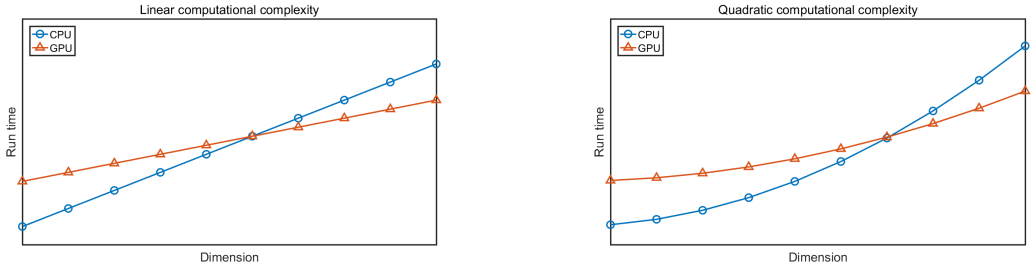


Figure 4-1: In this figure, computation B depends on computation A, hence these computations cannot be performed in parallel. In the case of pure parallelism a speed-up is very easily obtained. In the case of fine-grained parallelism a speed-up is only obtained for large-scale problems. In the case of coarse-grained parallelism a speed-up is obtained only for large-scale problems as well, but as can be seen coarse-grained parallelism is more effective than fine-grained parallelism.

As a result of these characteristics, the run time of sequential and parallel implementations of any numerical method have the expected profile as shown in Figure 4-2. For small dimensions the sequential implementation will always be faster than the parallel implementation, however there is a certain cross-over point where the parallel implementation becomes faster than the sequential implementation. It depends on the computational complexity whether the profile is linear, quadratic or of an even higher order.



(a) Computational complexity $\mathcal{O}(n)$ **(b)** Computational complexity $\mathcal{O}(n^2)$

Figure 4-2: Expected performance gain for parallel computing. For small dimensions, sequential computing will always be faster than parallel computing. For large dimensions, parallel computing may yield a speed-up.

4-3 GPU Architecture

GPUs were originally designed for rendering high resolution graphics at high frame rates. However, recently GPU capabilities have gained a lot of interest in the field of scientific computing [7]. Nowadays some GPUs are specifically designed for parallel computing in scientific applications [24, 25]. A GPU contains a certain number of multiprocessors, which each contain a certain numbers of cores. So-called threads, which are discussed in Section 4-4, can be executed on these cores. The CPU is referred to as the host and the GPU is referred to as the device [9]. Although double-precision GPUs exist, most GPUs work with single-precision [9, 24], which is why this project focusses on single-precision as well.

4-4 CUDA Programming

The compute unified device architecture (CUDA) platform is specifically developed by NVIDIA® [10, 11] in order to suit the GPU architecture and to gain the most out of its capabilities. A key element for this is the use of kernels.

Kernels

In CUDA, chunks of code can be gathered in so-called kernels, which are to be executed in parallel. A kernel can be compared to a function specified by the programmer, which requires certain inputs and returns a certain output. Where a function call causes the code in the function to be executed a single time, a kernel call may cause the code in the kernel

to be executed many times simultaneously, each time affecting a different memory location. In order to structure this massively parallel process of computations and communications, a certain thread hierarchy and memory hierarchy is used, which is illustrated by Figure 4-3.

4-4-1 Thread Hierarchy

Threads

Each instance of a kernel that is executed is referred to as a thread. Current GPUs are able to execute many thousands of threads simultaneously. Since a kernel is executed by many threads simultaneously, it is necessary to identify threads from within the kernel [10], making sure that each thread operates on its own part of the available data.

Blocks

Threads are organised in blocks, which can be one-, two- or three-dimensional. To identify each thread within a block, the `threadIdx` and `blockDim` variables are available. On current GPUs the maximum amount of threads per block is 1024 [10].

Grids

Blocks are organized in grids, which can also be one-, two- or three-dimensional. To identify each block within a grid, the `blockIdx` and `gridDim` variables are available. When calling a kernel, the block dimensions and the grid dimensions must be specified to make sure every thread has access to the correct memory location [10].

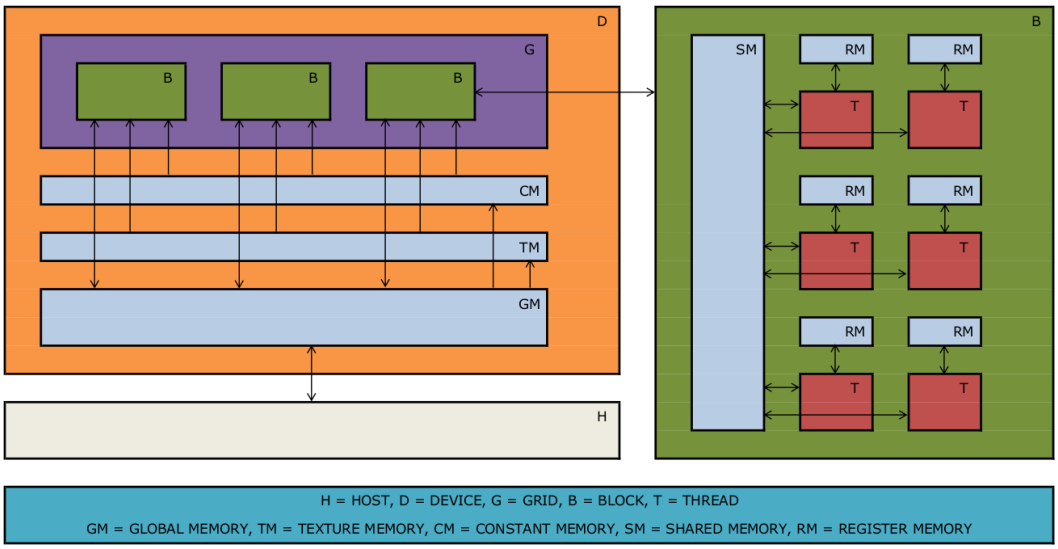


Figure 4-3: This CUDA thread and memory hierarchy example consists of a single grid containing three blocks, each containing six threads [10]. The right section is a more detailed representation of one of the blocks in the left section. The arrows represent communication possibilities.

Streams

The discussed hierarchy is designed to effectively perform parallel computations. When doing parallel computing, it is important that operations are synchronized at the right moments, in order to make sure that data are not being transferred to and from the same location at the same time. As a rule synchronization occurs before and after every CPU computation, every memory copy, every kernel call and every other CUDA command. Hence the only parallelism in standard CUDA code occurs either in the user-programmed kernels or in the pre-programmed CUDA commands. When more parallelism is desired, streams can be used in order to perform multiple kernels, CUDA commands or even memory copies at the same time. In this case it is important of course, to accurately track all operations and make sure that no conflicts occur between different communications and computations. In other words it is recommended to use streams for completely separate parts of an algorithm, that have no interdependency.

4-4-2 Memory Hierarchy

In order to allow threads, blocks, grids and streams to effectively deal with memory storage and communication, different memory locations on the GPU can be used. These memory locations are shown in Figure 4-3 as well.

Register memory

Threads have access to their own on-chip register memory. Because it is on-chip, register memory is very fast, moreover it has the same lifetime as the thread itself. Therefore, before computation, data need to be acquired from the shared memory or the global memory and after computation, data need to be transferred back to the shared memory or the global memory [10, 9].

Shared memory

Threads within a block can cooperate by sharing data through the on-chip shared memory. Because it is on-chip, shared memory is also very fast, moreover it has the same lifetime as the block itself. Therefore, before computation, data need to be acquired from the global memory and after computation, data need to be transferred back to the global memory [10].

Global memory

Global memory resides in the device memory and is accessible by all blocks and therefore by all threads as well. Since it is not on-chip, global memory has a much higher latency and much lower bandwidth than shared memory [10, 9].

Constant and texture memory

Constant and texture memory both reside in the device memory and are therefore just as slow as the global memory. They are accessible by all blocks and therefore by all threads as well, however they are read-only. In some specific cases, accessing data via the texture memory may have advantages over accessing data via the global memory [10].

Local memory

Every thread has some private local memory space assigned to it, which resides in the device memory. Local memory accesses only occur for some automatic variables or when the register memory is full. Since register memory is much faster than local memory, it is preferable to avoid the use of local memory [10].

Host memory

Since data transfer between host memory and device memory is very slow, it should be kept at a minimum level. This can be achieved for instance by moving more code from the host to the device, even if that means executing kernels with low parallelism. Intermediate data structures may be created in device memory, operated on by the device, and destroyed without ever being copied to host memory. Also, because of the overhead associated with each transfer, batching many small transfers into a single large transfer always performs better than making each transfer separately [10].

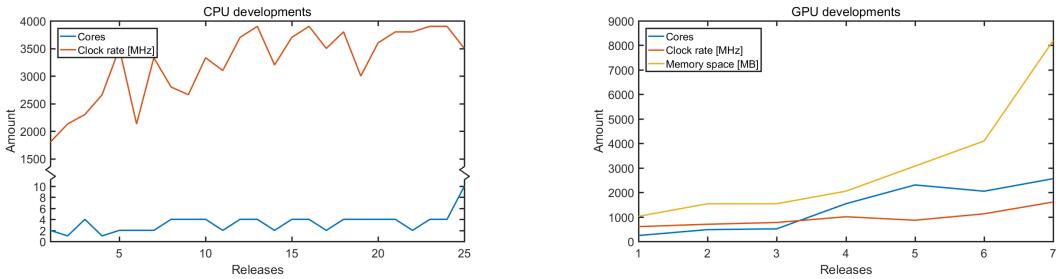
Naturally, there is a limit to the amount of storage space on the GPU. Since the device memory is the largest memory space, this is the limiting factor. Hence there is always a trade-off between avoiding communication between host and device and the amount of available storage space on the device.

4-5 Relevant Tools

There are many tools available for efficiently performing computations, both sequentially and in parallel. For performing the basic linear algebra subprograms (BLAS) a C library is available at http://www.netlib.org/blas called CBLAS and a CUDA library is available called CUBLAS [26]. Since BLAS libraries are highly optimized, it is always recommended to use these libraries rather than programming manual linear algebra routines. Another library that may be useful is DLIB, available at http://www.dlib.net. This C++ library consists of many useful tools, among which a package dedicated to optimization. Also CUDA has some libraries available for these purposes, for instance the CUSOLVER library consists of some linear optimization tools [27]. Finally pseudo-random number generation is an important aspect of the DONE algorithm. The CURAND library is able to generate large amounts of pseudo-random numbers using parallel computing [28].

4-6 Recent Developments

The reason that parallel computing has so much more promise for the future than sequential computing, is that CPU clock rate developments seem to be reaching a limit [7, 8] whereas GPU developments are still in full progress [24, 25]. Moreover CPU capacities can only be improved by improving the clock rate whereas GPU capacities can be improved by improving the clock rate and, more importantly, also by increasing the amount of cores. Figure 4-4a shows the clock rate improvements of Intel® CPUs over various releases and also the amount of cores. As can be seen, in order to improve CPU capacities, the multi-core approach is applied as well. Figure 4-4b shows the clock rate improvements of NVIDIA® GPUs over various releases and also the amount of cores and memory space.



(a) CPU developments **(b)** GPU developments

Figure 4-4: Recent developments in CPU and GPU production. The releases that are referred to are listed in Table 4-1.

As can be seen GPU clock rates are still below CPU clock rates, but as discussed in Section 4-2 it is not the clock rate that makes a GPU such a powerful tool but the massive parallelism that is a result of the large amount of cores. Furthermore the CPU clock rate trend seems to be that less improvement is made every year [7]. Where CPU clock rates used to improve with 52% per year, recently this has diminished to 22% per year [8]. As can be seen in Figure 4-4 the CPU clock rate indeed seems to be reaching its limit whereas the GPU clock rate and the amount of cores are both still increasing. Physically, it may be expectable that all clock rates should eventually have an upper limit, whereas the amount of cores can be extended unlimitedly. This is why parallel computing is so much more promising than sequential computing. Moreover as can be seen in Figure 4-4b also the available memory on GPUs increases very rapidly, which allows the large amounts of cores to be used to their full capacity. Table 4-1 lists which releases are referred to by Figure 4-4 and where the information originates from.

4-7 Parallel Computing Criteria

Based on the theory discussed in this chapter, several criteria are established which measure the suitability of a numerical method for parallel computing. Numerical methods that are candidates for parallel computing should be judged based on the following criteria:

- Computational complexity
- Degree of parallelism
- Required memory storage

This should help deciding which of the methods are suitable for further research and possible implementation and which of them are not. It may be interesting to find out if a trade-off between these criteria may lead to an unexpected best performing method.

CPU releases	**GPU releases**
Legacy Intel® Core™2	NVIDIA® GeForce® GTX 280
Legacy Intel® Celeron®	NVIDIA® GeForce® GTX 480
Intel® Celeron®	NVIDIA® GeForce® GTX 580
Legacy Intel® Pentium®	NVIDIA® GeForce® GTX 680
Intel® Pentium®	NVIDIA® GeForce® GTX 780
Intel® Atom™	NVIDIA® GeForce® GTX 980
Previous Generation Intel® Core™ i3, i5, i7	NVIDIA® GeForce® GTX 1080
Previous Generation Intel® Core™ i7 Extreme	
2nd Generation Intel® Core™ i3, i5, i7	
3rd Generation Intel® Core™ i3, i5, i7	
4th Generation Intel® Core™ i3, i5, i7	
5th Generation Intel® Core™ i5, i7	
6th Generation Intel® Core™ i3, i5, i7	
High End Desktop Processors	

Table 4-1: Recent CPU and GPU releases by Intel® and NVIDIA® respectively. The CPU information originates from http://ark.intel.com and the GPU information originates from http://www.geforce.com/hardware/desktop-gpus. For each CPU release the version at the top of the list is used in Figure 4-4a.

Chapter 5

RFE Fitting

This chapter discusses the RFE fitting step of the DONE algorithm, shown in Figure 5-1. Several methods are presented to solve the regularized linear least-squares problem (3-6, 3-7) which is established in Section 3-2 and is denoted by

$$\boldsymbol{w}_i = \underset{\boldsymbol{w}}{\operatorname{argmin}}\, J_i(\boldsymbol{w}), \tag{5-1}$$

$$J_i(\boldsymbol{w}) = ||\boldsymbol{y}_i - \boldsymbol{A}_i\boldsymbol{w}||_2^2 + \lambda\,||\boldsymbol{w}||_2^2\,, \tag{5-2}$$

where $\boldsymbol{A}_i \in \mathbb{R}^{i\times m}$ and $\boldsymbol{y}_i \in \mathbb{R}^i$ are augmented every i^{th} cycle of the DONE algorithm with a new set of measurements $\{\boldsymbol{x}_i \in \mathcal{X}, y_i \in \mathbb{R}\}$. The goal of the RFE fitting step is to determine the optimal weight vector $\boldsymbol{w}_i$ in order to establish the surrogate function $\hat{f}_i$ that is to be optimized in the optimization step.

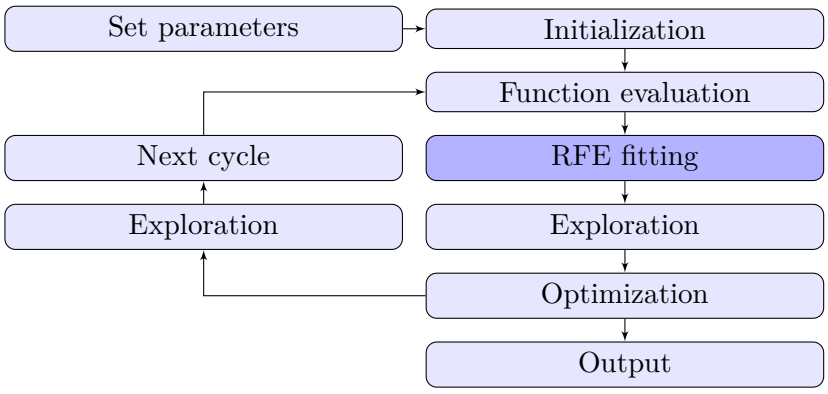


Figure 5-1: In this flowchart of the DONE algorithm, the RFE fitting step is dark shaded. For a more elaborate flowchart, see Figure 3-1

5-1 Least-Squares Strategy

5-1-1 Optimization-Based Methods

An important remark is that the regularized linear least-squares cost function (5-2) can also be denoted by

$$\begin{aligned} J_i(\boldsymbol{w}) &= \boldsymbol{w}^T(\boldsymbol{A}_i^T\boldsymbol{A}_i + \lambda \boldsymbol{I}_m)\boldsymbol{w} - 2\boldsymbol{w}^T\boldsymbol{A}_i^T\boldsymbol{y}_i + \boldsymbol{y}_i^T\boldsymbol{y}_i \\ &= \boldsymbol{w}^T\boldsymbol{\Phi}_i\boldsymbol{w} - 2\boldsymbol{w}^T\boldsymbol{s}_i + \boldsymbol{y}_i^T\boldsymbol{y}_i, \end{aligned} \tag{5-3}$$

from which it immediately becomes clear that it is a quadratic function. The gradient and Hessian are denoted by

$$\nabla_{\boldsymbol{w}} J_i(\boldsymbol{w}) = 2\boldsymbol{\Phi}_i\boldsymbol{w} - 2\boldsymbol{s}_i, \tag{5-4}$$

$$\nabla^2_{\boldsymbol{w}} J_i(\boldsymbol{w}) = 2\boldsymbol{\Phi}_i, \tag{5-5}$$

where the Hessian is symmetric and positive-definite, indicating that the cost function (5-3) is not only quadratic, but also convex. Hence the optimal weight vector $\boldsymbol{w}_i$ can easily be computed using the optimization methods discussed in Chapter 6. The methods which are suitable for this problem are:

- Gradient descent (GD)
- Nesterov's accelerated gradient descent (NGD)
- Linear Fletcher-Reeves (LFR)
- Nonlinear Fletcher-Reeves (NFR)
- Davidon-Fletcher-Powell (DFP)
- Broyden-Fletcher-Goldfarb-Shanno (BFGS)
- Limited memory Broyden-Fletcher-Goldfarb-Shanno (LBFGS)

The methods that are not in this list are Newton (N) and Levenberg-Marquardt (LM). It seems a bit redundant to use methods that perform Hessian inversions iteratively in order to solve a problem which only requires a single Hessian inversion to obtain the solution (5-6).

5-1-2 Recursive Methods

The solution to the regularized linear least-squares problem (5-1, 5-2) is denoted by

$$\boldsymbol{w}_i = (\boldsymbol{A}_i^T\boldsymbol{A}_i + \lambda \boldsymbol{I}_m)^{-1}\boldsymbol{A}_i^T\boldsymbol{y}_i = \boldsymbol{\Phi}_i^{-1}\boldsymbol{s}_i, \tag{5-6}$$

which corresponds to solving a set of linear equations. Since $\boldsymbol{y}_i \in \mathbb{R}^i$ and $\boldsymbol{A}_i \in \mathbb{R}^{i\times m}$ do not completely change every cycle, but are merely augmented using

$$\boldsymbol{y}_i = \begin{bmatrix} \boldsymbol{y}_{i-1}^T & y_i \end{bmatrix}^T, \quad \boldsymbol{A}_i = \begin{bmatrix} \boldsymbol{A}_{i-1} \\ \boldsymbol{a}_i^T \end{bmatrix},$$

it is not necessary to solve the full regularized linear least-squares problem every cycle. The update relations are denoted by

$$\boldsymbol{\Phi}_i = \boldsymbol{\Phi}_{i-1} + \boldsymbol{a}_i \boldsymbol{a}_i^T, \tag{5-7}$$

$$\boldsymbol{s}_i = \boldsymbol{s}_{i-1} + \boldsymbol{a}_i y_i, \tag{5-8}$$

where $\boldsymbol{\Phi}_i \in \mathbb{R}^{m \times m}$ and $\boldsymbol{s}_i \in \mathbb{R}^m$ do not change in size when an extra measurement is added [14]. Using these update relations the solution (5-6) can be computed every cycle in a very efficient, recursive fashion.

Recursive least-squares method

By defining $\boldsymbol{P}_i = \boldsymbol{\Phi}_i^{-1}$ and $\boldsymbol{g}_i = \boldsymbol{\Phi}_i^{-1} \boldsymbol{a}_i$ and starting off with

$$\boldsymbol{P}_0 = \lambda^{-1} \boldsymbol{I}_m, \quad \boldsymbol{w}_0 = \boldsymbol{0}_m,$$

the recursive least-squares (RLS) solution can be computed using

$$\gamma_i = (1 + \boldsymbol{a}_i^T \boldsymbol{P}_{i-1} \boldsymbol{a}_i)^{-1}, \tag{5-9}$$

$$\boldsymbol{g}_i = \gamma_i \boldsymbol{P}_{i-1} \boldsymbol{a}_i, \tag{5-10}$$

$$\boldsymbol{P}_i = \boldsymbol{P}_{i-1} - \gamma_i^{-1} \boldsymbol{g}_i \boldsymbol{g}_i^T, \tag{5-11}$$

$$\boldsymbol{w}_i = \boldsymbol{w}_{i-1} - \boldsymbol{g}_i \boldsymbol{a}_i^T \boldsymbol{w}_{i-1} + \boldsymbol{g}_i y_i, \tag{5-12}$$

which allows quick computation of the optimal weight vector $\boldsymbol{w}_i$ without having to solve the full linear least-squares problem (5-1, 5-2) and without having to compute any matrix inverse or solve a set of linear equations [14].

In applications where temporal aspects play a big role, it may be useful to implement a so-called forgetting factor, which puts a decaying weight on previous measurements. For the DONE algorithm this may be useful if older measurements become out of scope and are no longer representative for the current situation.

Inverse QR method

The recursive least-squares computations (5-9, 5-10, 5-11) can also be expressed as

$$\begin{bmatrix} 1 & \boldsymbol{a}_i^T \boldsymbol{P}_{i-1}^{\frac{1}{2}} \\ \boldsymbol{0}_m & \boldsymbol{P}_{i-1}^{\frac{1}{2}} \end{bmatrix} \begin{bmatrix} 1 & \boldsymbol{0}_m^T \\ \boldsymbol{P}_{i-1}^{\frac{T}{2}} \boldsymbol{a}_i & \boldsymbol{P}_{i-1}^{\frac{T}{2}} \end{bmatrix} = \begin{bmatrix} \gamma_i^{-\frac{1}{2}} & \boldsymbol{0}_m^T \\ \boldsymbol{g}_i \gamma_i^{-\frac{1}{2}} & \boldsymbol{P}_i^{\frac{1}{2}} \end{bmatrix} \begin{bmatrix} \gamma_i^{-\frac{1}{2}} & \boldsymbol{g}_i^T \gamma_i^{-\frac{1}{2}} \\ \boldsymbol{0}_m & \boldsymbol{P}_i^{\frac{T}{2}} \end{bmatrix}, \tag{5-13}$$

which can be interpreted as the orthogonal matrix rotation

$$\begin{bmatrix} 1 & \boldsymbol{a}_i^T \boldsymbol{P}_{i-1}^{\frac{1}{2}} \\ \boldsymbol{0}_m & \boldsymbol{P}_{i-1}^{\frac{1}{2}} \end{bmatrix} \boldsymbol{\Theta} = \begin{bmatrix} \gamma_i^{-\frac{1}{2}} & \boldsymbol{0}_m^T \\ \boldsymbol{g}_i \gamma_i^{-\frac{1}{2}} & \boldsymbol{P}_i^{\frac{1}{2}} \end{bmatrix}, \tag{5-14}$$

where $\boldsymbol{\Theta} \in \mathbb{R}^{(m+1)\times(m+1)}$ is an orthogonal rotation matrix which transforms the upper triangular matrix on the left into the lower triangular matrix on the right. This is referred to as the inverse QR (IQR) method [14]. Note that the upper triangular matrix only contains entries which are known at the i^{th} cycle, whereas the lower triangular matrix contains entries which are required to compute $\boldsymbol{w}_i$ using (5-12). An advantage of this method is that it does not matter how the orthogonal rotation (5-14) is performed, this can be done in many ways [14, 29]. The rotation matrix $\boldsymbol{\Theta}$ can for instance be computed by performing a QR decomposition [1, 29] or the rotation can be performed using Givens rotations, without even computing an explicit expression for $\boldsymbol{\Theta}$ [14, 29].

Since the (lower triangular) square root factor $\boldsymbol{P}_i^{\frac{1}{2}} \in \mathbb{R}^{m\times m}$ is used, the matrix $\boldsymbol{P}_i = \boldsymbol{P}_i^{\frac{1}{2}} \boldsymbol{P}_i^{\frac{T}{2}}$ will always be positive-definite. This makes the inverse QR method more numerically stable than the standard recursive least-squares method, where $\boldsymbol{P}_i \in \mathbb{R}^{m\times m}$ can lose its positive-definiteness due to numerical inaccuracies [14].

QR method

The QR method is very similar to the IQR method, the difference is that $\boldsymbol{\Phi}_i$ is computed instead of $\boldsymbol{P}_i$. This is desirable when λ becomes very large, causing the initial matrix $\boldsymbol{P}_0$ to become very small. The QR method uses the matrix transformation denoted by

$$\begin{bmatrix} \boldsymbol{\Phi}_{i-1}^{\frac{1}{2}} & \boldsymbol{a}_i \\ \boldsymbol{w}_{i-1}^T \boldsymbol{\Phi}_{i-1}^{\frac{1}{2}} & y_i \\ \boldsymbol{0}_m^T & 1 \end{bmatrix} \boldsymbol{\Theta} = \begin{bmatrix} \boldsymbol{\Phi}_i^{\frac{1}{2}} & \boldsymbol{0}_m \\ \boldsymbol{w}_i^T \boldsymbol{\Phi}_i^{\frac{1}{2}} & (y_i - \boldsymbol{a}_i^T \boldsymbol{w}_i)\gamma_i^{\frac{1}{2}} \\ \boldsymbol{a}_i^T \boldsymbol{\Phi}_i^{-\frac{T}{2}} & \gamma_i^{\frac{1}{2}} \end{bmatrix}, \tag{5-15}$$

where all entries of the lower triangular matrix on the left are known at the i^{th} cycle and only the upper left entry and the middle left entry of the upper triangular matrix on the right are required to compute $\boldsymbol{w}_i$. A disadvantage of the QR methods is that a lower triangular set of linear equations needs to be solved in order to compute the solution whereas the inverse QR method only requires straightforward matrix and vector computations. This problem can be overcome by extending the matrices used in the transformation (5-15), however this may lead to numerical complications [14]. The orthogonal rotation can be performed in a similar fashion to the inverse QR method.

5-2 Comparison

As discussed in Section 5-1-1 it is not useful to use the Newton method or the Levenberg-Marquardt method in this case, therefore these methods are not taken into account for the comparison. Furthermore the conversion of the measurements

$$\boldsymbol{a}_i = \cos(\boldsymbol{\Omega}^T \boldsymbol{x}_i + \boldsymbol{b}) \tag{5-16}$$

from $\boldsymbol{x}_i \in \mathcal{X}$ to $\boldsymbol{a}_i \in \mathcal{A}$, where $\mathcal{A} \subseteq \mathbb{R}^m$, is required for every RFE fitting method, hence this is not taken into account for the comparison either. Table 5-1 lists the ranking of all discussed RFE fitting methods based on the parallel computing criteria established in Section 4-7. The primary computations are the computations that are inherent to the methods and the secondary computations are the computations that are required to compute gradient values (5-4). Note that the RLS, IQR and QR methods do not require any secondary computations.

Method	**Primary**	**Secondary**
RLS	$\mathcal{O}(m^2)$	
IQR	$\mathcal{O}(m^2)$	
QR	$\mathcal{O}(m^2)$	
GD	$\mathcal{O}(m)$	$\mathcal{O}(m^2)$
NGD	$\mathcal{O}(m)$	$\mathcal{O}(m^2)$
NFR	$\mathcal{O}(m)$	$\mathcal{O}(m^2)$
LFR	$\mathcal{O}(m^2)$	$\mathcal{O}(m^2)$
DFP	$\mathcal{O}(m^2)$	$\mathcal{O}(m^2)$
BFGS	$\mathcal{O}(m^2)$	$\mathcal{O}(m^2)$
LBFGS	$\mathcal{O}(m^2)$	$\mathcal{O}(m^2)$

(a) Computational complexity

Method	**Primary**	**Secondary**
RLS	+++	
IQR	+++	
QR	+	
GD	+++++	+++
NGD	+++++	+++
NFR	+++	+++
LFR	+++	+++
DFP	+++	+++
BFGS	+++	+++
LBFGS	+++	+++

(b) Degree of parallelism

Method	**Primary**	**Secondary**
RLS	$\mathcal{O}(m^2)$	
IQR	$\mathcal{O}(m^2)$	
QR	$\mathcal{O}(m^2)$	
GD	$\mathcal{O}(m)$	$\mathcal{O}(m^2)$
NGD	$\mathcal{O}(m)$	$\mathcal{O}(m^2)$
NFR	$\mathcal{O}(m)$	$\mathcal{O}(m^2)$
LFR	$\mathcal{O}(m)$	$\mathcal{O}(m^2)$
DFP	$\mathcal{O}(m^2)$	$\mathcal{O}(m^2)$
BFGS	$\mathcal{O}(m^2)$	$\mathcal{O}(m^2)$
LBFGS	$\mathcal{O}(m)$	$\mathcal{O}(m^2)$

(c) Memory storage

Method	**Suitability**
RLS	Yes
IQR	Yes
QR	No
GD	Yes
NGD	Yes
NFR	Yes
LFR	Yes
DFP	No
BFGS	No
LBFGS	No

(d) Overall suitability

Table 5-1: RFE fitting methods judged based on parallel computing criteria. The primary columns indicate the inherent properties of the methods, the secondary columns indicate the properties that are imposed by gradient computations (5-4).

Computational complexity

Primarily all methods except for the first-order optimization-based methods have computational complexity $\mathcal{O}(m^2)$. However due to the gradient computation (5-4) all optimization methods (secondarily) have computational complexity $\mathcal{O}(m^2)$, so although first-order optimization-based methods primarily appear to be the better choice, secondarily this is not the case. Note that the recursive methods do not require any secondary computations.

Degree of parallelism

Since the GD and NGD methods solely require pure vector computations, they are fully parallelizable and thus have a very high primary degree of parallelism. Since the second-order optimization-based methods require operations like (4-4), they have a medium primary degree of parallelism. Since the QR method requires solving a set of linear equations, it has a low primary degree of parallelism. The gradient computation (5-4) gives the optimization-based methods a medium secondary degree of parallelism. For the recursive methods the secondary degree of parallelism is not applicable.

Memory storage

Since the first-order optimization-based methods only need to store vectors and no matrices, they primarily only require $\mathcal{O}(m)$ memory storage. Due to its special structure, this also applies to the second-order optimization-based method LBFGS. All other methods primarily require $\mathcal{O}(m^2)$ memory storage. Since for the optimization-based methods $\mathbf{\Phi}_i \in \mathbb{R}^{m \times m}$ needs to be stored in order to compute the gradient (5-4), secondarily $\mathcal{O}(m^2)$ memory storage is required anyway. For the recursive methods, the secondary memory storage is not applicable. Note that for all methods $\mathbf{\Omega} \in \mathbb{R}^{n \times m}$ is needed to perform the domain conversion (5-16) and although this is not taken into account in this comparison, for this $\mathcal{O}(mn)$ memory storage is required no matter what.

Overall suitability

The QR method is dismissed based on the fact that it has a lower degree of parallelism than its companions IQR and RLS. The DFP and BFGS methods are dismissed based on the fact that they require both primary and secondary memory storage $\mathcal{O}(m^2)$. Moreover it is rather redundant to use a second-order optimization-based method which approximates an inverse Hessian $\boldsymbol{B}_k$ while an expression for the real Hessian (5-5) is already available in the form of $\mathbf{\Phi}_i$. Based on this argument, plus the fact that it is rather complicated to implement, LBFGS is dismissed as well.

Chapter 6

Optimization

This chapter discusses the optimization step of the DONE algorithm, which is shown in Figure 6-1. Several methods are presented to solve the optimization problem (3-10, 3-11), which is established in Section 3-3 and is denoted by

$$\boldsymbol{x}_{i+1} = \underset{\boldsymbol{x}}{\operatorname{argmin}}\, \hat{f}_i(\boldsymbol{x}) \quad \text{s.t.} \quad \boldsymbol{x} \in \mathcal{X} \tag{6-1}$$

$$\hat{f}_i(\boldsymbol{x}) = \boldsymbol{w}_i^T \cos(\boldsymbol{\Omega}^T \boldsymbol{x} + \boldsymbol{b}) \tag{6-2}$$

where $\boldsymbol{\Omega} \in \mathbb{R}^{n \times m}$ and $\boldsymbol{b} \in \mathbb{R}^m$ contain the randomly selected frequencies and phases, $\boldsymbol{w}_i \in \mathbb{R}^m$ is the optimal weight vector constructed in the RFE fitting step and the feasible region $\mathcal{X}$ is defined by a lower bound $\boldsymbol{x}_{lb} \in \mathbb{R}^n$ and an upper bound $\boldsymbol{x}_{ub} \in \mathbb{R}^n$.

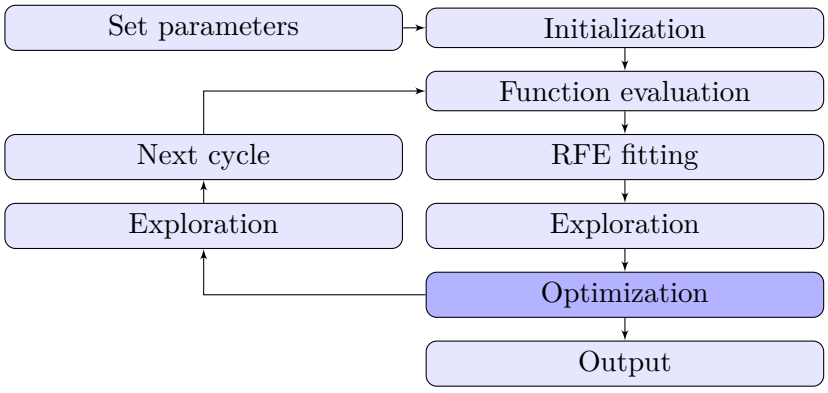


Figure 6-1: In this flowchart of the DONE algorithm, the optimization step is dark shaded. For a more elaborate flowchart, see Figure 3-1

6-1 Optimization Strategy

Multidimensional optimization problems can be solved using two approaches, the line search approach and the trust region approach [6]. From here on $k \in \mathbb{Z}$ is the iteration counter.

6-1-1 Line Search Approach

In the line search approach, each iteration a search direction $\boldsymbol{p}_k \in \mathbb{R}^n$ is selected along which is searched for a lower function value. This procedure simplifies the problem of optimizing the multidimensional function $\hat{f}_i : \mathbb{R}^n \mapsto \mathbb{R}$ to a series of evaluations of the one-dimensional line search function $\phi_k : \mathbb{R} \mapsto \mathbb{R}$, which is denoted by

$$\phi_k(\alpha_k) = \hat{f}_i(\boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k), \tag{6-3}$$

where α_k is the size of the step taken along the search direction $\boldsymbol{p}_k$. The best strategy is of course to search for the optimal step size

$$\alpha_k = \underset{\alpha}{\operatorname{argmin}}\, \phi_k(\alpha). \tag{6-4}$$

For some functions, like quadratic functions, this can easily be accomplished, however for most nonlinear functions this may be a time-consuming process. Luckily it suffices to select a step size which leads to a sufficient decrease in function value [6].

6-1-2 Trust Region Approach

In the trust region approach, a model function $\vartheta_k : \mathbb{R}^n \mapsto \mathbb{R}$ is constructed whose behaviour near the current point $\boldsymbol{x}_k$ is similar to that of the actual function $\hat{f}_i : \mathbb{R}^n \mapsto \mathbb{R}$. The model function ϑ_k is often chosen to be a quadratic function, of which the exact optimum can be computed analytically. This model is however only valid in a certain region around the current point $\boldsymbol{x}_k$, which is called the trust region. If the optimum of the model function does not produce a sufficient decrease in the actual function value, the trust region is probably too large and is therefore shrunk, after which the procedure is repeated. When a sufficient decrease in function value is reached, a new model function ϑ_{k+1} is constructed around the new point $\boldsymbol{x}_{k+1}$ [6].

6-1-3 Comparison

One may point out that the trust region approach can be compared to the DONE algorithm itself, except that the model function is in this case not quadratic and no derivatives can be used to construct it. Hence it would be rather ineffective to use a trust region approach to optimize the surrogate function $\hat{f}_i : \mathbb{R}^n \mapsto \mathbb{R}$, which is already a model of the objective function $f : \mathbb{R}^n \mapsto \mathbb{R}$. Moreover the trust region approach requires the optimum of the model function to be computed analytically, which comes down to solving a set of linear equations. For large values of n this becomes a very large problem which may be very time-consuming.

A benefit of the line search approach is that, no matter how large n becomes, the line search function ϕ_k remains one-dimensional. Moreover, no exact minima need to computed whatsoever. Even with a step size that only leads to little decrease of the function value, the procedure will eventually converge to the optimum. Hence the line search approach allows the problem at hand (6-1, 6-2) to be solved in a much more efficient fashion than the trust region approach. For this reason, the trust region approach will not be considered any further.

6-2 Step Size Selection

An important aspect of the line search approach, one which has a very large influence on the convergence rate of the optimization method, is the selection of the step size α_k. As discussed in Section 6-1-1, it is preferable to select the step size such that the line search function ϕ_k is exactly minimized. However since this may be very time-consuming, less efficient step sizes may be considered as well.

6-2-1 Fixed Step Size

The most simple approach is to select a fixed step size α in advance. This step size is then used throughout the entire optimization procedure. The benefit of this approach is that no extra computations are required during the iterations. Function values do not even need to be computed at all. The only required operation is

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha \boldsymbol{p}_k, \tag{6-5}$$

after which a new search direction $\boldsymbol{p}_k$ can be chosen and the next iteration can be started. A disadvantage of this approach is that the step size may either be too large, resulting in an increase in function value and thus destroying the convergence, or too small, resulting in a series of many very small steps where one larger step would have been sufficient.

6-2-2 Wolfe Conditions

In order to deal with the problems that arise when the step size is not properly chosen, the Wolfe conditions can be taken into account [6]. These conditions may ensure that the step size is chosen such that convergence is guaranteed within an acceptable amount of iterations. The Wolfe conditions are denoted by

$$\hat{f}_i(\boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k) \leq \hat{f}_i(\boldsymbol{x}_k) + c_1 \alpha_k \boldsymbol{p}_k^T \nabla_{\boldsymbol{x}} \hat{f}_i(\boldsymbol{x}_k), \tag{6-6}$$

$$\boldsymbol{p}_k^T \nabla_{\boldsymbol{x}} \hat{f}_i(\boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k) \geq c_2 \boldsymbol{p}_k^T \nabla_{\boldsymbol{x}} \hat{f}_i(\boldsymbol{x}_k), \tag{6-7}$$

where $c_1 \in [0, 1]$ and $c_2 \in [0, 1]$ are predefined constants. The first condition is called the sufficient decrease condition and also sometimes the Armijo condition [6, 30], the second condition is called the curvature condition [6]. In order to acquire a better understanding of the Wolfe conditions, they can be transformed into

$$\phi_k(\alpha_k) \leq \phi_k(0) + c_1\alpha_k\phi_k'(0), \tag{6-8}$$
$$\phi_k'(\alpha_k) \geq c_2\phi_k'(0), \tag{6-9}$$

where ϕ_k is the line search function discussed in Section 6-1-1 and ϕ_k' is its derivative with respect to α_k. Hence the sufficient decrease condition (6-6, 6-8) dismisses too large step sizes, as shown in Figure 6-2, and the curvature condition (6-7, 6-9) dismisses too small step sizes, as shown in Figure 6-3. Note that in these examples $c_1 = c_2$.

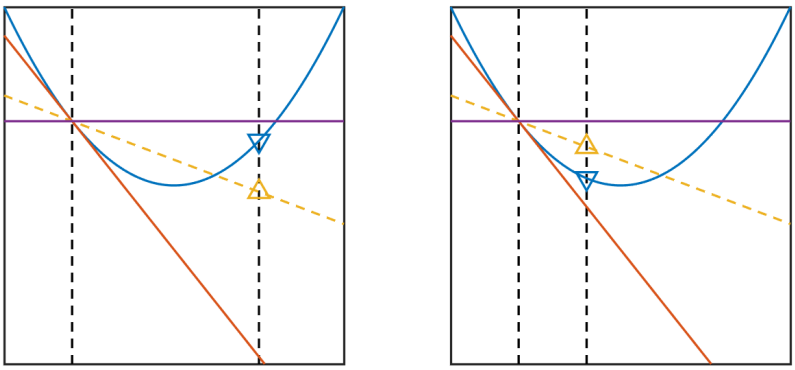


Figure 6-2: The blue parabola represents ϕ_k, the left vertical dashed black line represents $\alpha_k = 0$ and the right vertical dashed black line represents any chosen $\alpha_k > 0$. For too large values of α_k (left subfigure) the yellow upward pointing triangle, which represents $\phi_k(0) + c_1\alpha_k\phi_k'(0)$, lies below the blue downward pointing triangle, which represents $\phi_k(\alpha_k)$. For sufficiently small values of α_k (right subfigure) the sufficient decrease condition (6-6, 6-8) is satisfied.

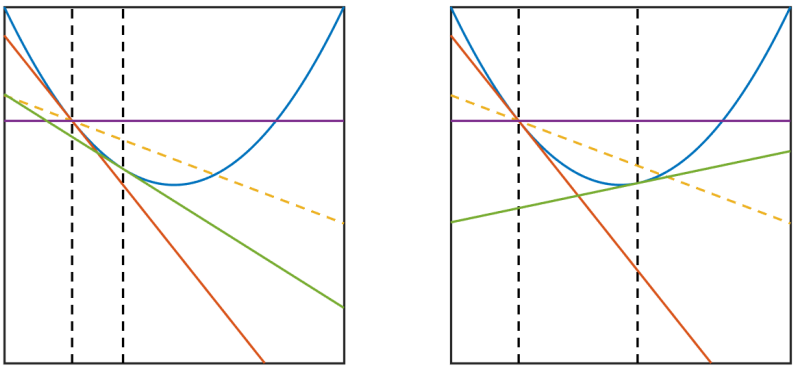


Figure 6-3: The blue parabola represents ϕ_k, the left vertical dashed black line represents $\alpha_k = 0$ and the right vertical dashed black line represents any chosen $\alpha_k > 0$. For too small values of α_k (left subfigure) the slope $\phi_k'(\alpha_k)$ of the green tangent line is less than the slope $c_2\phi_k'(0)$ of the yellow dashed line. For sufficiently large values of α_k (right subfigure) the curvature condition (6-7, 6-9) is satisfied. Note that some of the slopes are negative in this example.

As stated the sufficient decrease condition is also called the Armijo condition. Remarkable is that for the Armijo condition, a value of $c_1 = 0.5$ is suggested [30], whereas for the Wolfe conditions, a value of $c_1 = 10^{-4}$ is suggested [6]. Of these values $c_1 = 0.5$ is the strictest, since it requires the largest function value decrease.

6-2-3 Backtracking

Backtracking is a method to select a satisfactory step size. The approach is to start with a rather large step size and gradually decrease it until the Wolfe conditions are satisfied. A benefit of this procedure is that the curvature condition (6-7, 6-9) can be omitted since the satisfactory step size is approached from above [6].

Classic approach

The classic approach is to decrease the step size simply by multiplying it with a predefined factor $\tau \in [0, 1]$ until the sufficient decrease condition (6-6, 6-8) is satisfied [6].

Extrapolation approach

The extrapolation approach decreases the step size by constructing a one-dimensional quadratic model of ϕ_k using the available values of $\phi_k(0)$, $\phi_k'(0)$ and $\phi_k(\alpha_k)$. The minimizer of this quadratic model is taken as the new value for α_k, after which a one-dimensional cubic model can be constructed. By memorizing two subsequent values of α_k at a time, new models can be constructed and minimized iteratively, ultimately converging to a value of α_k which satisfies the sufficient decrease condition [6].

6-3 Search Directions

For the optimization step, first-order and second-order search directions are investigated. Second-order search directions are more complicated, but they have better convergence properties. Appendix C discusses how convergence rates can be calculated.

6-3-1 First-Order Methods

For optimization problems, derivatives provide very useful information. First-order methods use the gradient, which contains information about the direction in which the function increases the most. From here on, the gradient at the point $\boldsymbol{x}_k$ is denoted by $\boldsymbol{r}_k = \nabla_{\boldsymbol{x}} \hat{f}_i(\boldsymbol{x}_k)$.

Gradient descent method

The gradient descent (GD) method searches in the direction of the negative gradient, which is also called the steepest descent direction, denoted by

$$\boldsymbol{p}_k = -\boldsymbol{r}_k. \tag{6-10}$$

A characteristic of the gradient descent method is that, if the optimal step size is used instead of a predefined step size, successive search directions are related by

$$\boldsymbol{p}_k^T \boldsymbol{p}_{k+1} = 0, \tag{6-11}$$

which indicates that they are perpendicular. If α_k is chosen properly, the gradient descent method converges to the nearest local optimum within a finite amount of iterations [6], moreover it does so with the sublinear convergence rate $\mathcal{O}(\frac{1}{k})$ [31, 32]. Figure 6-4a illustrates the gradient descent method.

Nesterov's accelerated gradient descent method

A downside of the gradient descent method is that while the optimum is approached, and thus the size of the gradient decreases, if α_k is fixed, the effective steps also become smaller, which drastically increases the amount of required iterations. In order to overcome this problem, Nesterov's accelerated gradient descent (NGD) method propagates not only the input vector $\boldsymbol{x}_k$ but also a momentum vector $\boldsymbol{v}_k$, which follow the relations

$$\boldsymbol{v}_{k+1} = \boldsymbol{x}_k - \alpha \boldsymbol{r}_k, \tag{6-12}$$

$$\boldsymbol{x}_{k+1} = (1 - \gamma_k)\boldsymbol{v}_{k+1} + \gamma_k \boldsymbol{v}_k, \tag{6-13}$$

where the parameter γ_k is computed using

$$\tau_{k+1} = \frac{1}{2} + \frac{1}{2}\sqrt{1 + 4\tau_k^2}, \tag{6-14}$$

$$\gamma_k = \tau_{k+1}^{-1}(1 - \tau_k), \tag{6-15}$$

which starts off with $\tau_0 = 0$ [31]. A requirement for Nesterov's accelerated gradient descent method is that α is the inverse of a Lipschitz constant of the gradient of the surrogate function (3-12) [31, 33]. The path followed may appear less efficient than the path followed by the gradient descent method, however it requires much less iterations. Nesterov's accelerated gradient descent method excels if the optimum is located in a very shallow area.

If α is chosen properly, Nesterov's accelerated gradient descent method converges to the nearest local optimum within a finite amount of iterations, moreover it does so with the sublinear convergence rate $\mathcal{O}(\frac{1}{k^2})$ [31, 32, 34]. Figure 6-4b illustrates Nesterov's accelerated gradient descent method. Appendix D discusses an interpretation of how Nesterov's accelerated gradient descent method can be derived.

Linear Fletcher-Reeves method

The Fletcher-Reeves method is a conjugate gradient method. Conjugate gradient methods were initially developed for solving linear sets of equations [35], which results in a quadratic objective function with a symmetric and positive-definite Hessian $\boldsymbol{H} \in \mathbb{R}^{n \times n}$. This is demonstrated in Section (5-1-1). Where the gradient descent method invokes perpendicular search directions, the conjugate gradient method imposes

$$\boldsymbol{p}_k^T \boldsymbol{H} \boldsymbol{p}_{k+1} = 0, \tag{6-16}$$

which indicates that successive search directions are conjugate directions. Conjugate directions can be computed beforehand by for instance computing the eigenvectors of $\boldsymbol{H}$. However, the Fletcher-Reeves method is capable of computing the conjugate search directions iteratively. Starting off with the negative gradient as search direction, the next search direction can be computed using

$$\boldsymbol{p}_{k+1} = -\boldsymbol{r}_{k+1} + \frac{\boldsymbol{r}_{k+1}^T \boldsymbol{r}_{k+1}}{\boldsymbol{r}_k^T \boldsymbol{r}_k} \boldsymbol{p}_k, \tag{6-17}$$

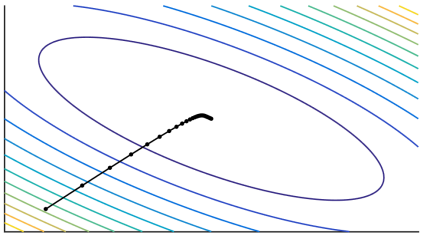
which only requires the current search direction and the current and next gradient. For quadratic objective functions an explicit expression for the optimal step size is denoted by

$$\alpha_k = -\frac{\boldsymbol{r}_k^T \boldsymbol{r}_k}{\boldsymbol{p}_k^T \boldsymbol{H} \boldsymbol{p}_k}. \tag{6-18}$$

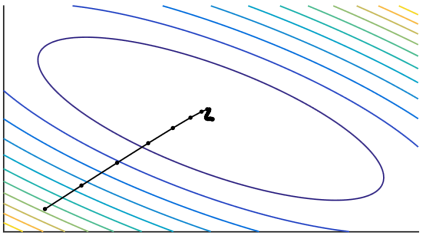
If α_k is computed properly, the linear Fletcher-Reeves method converges to the global optimum within n iterations, moreover it does so with a quadratic convergence rate [6]. Figure 6-4d illustrates the linear Fletcher-Reeves method.

Nonlinear Fletcher-Reeves method

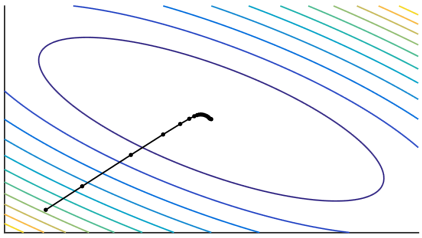
The linear Fletcher-Reeves can also be extended to nonlinear objective functions. In this case the step size α_k can be selected beforehand and no expression for the Hessian is required, confirming the first-order character of the Fletcher-Reeves method. If α_k is chosen properly, the nonlinear Fletcher-Reeves (NFR) method converges to the nearest local optimum within a finite amount of iterations [6]. For quadratic convex objective functions the nonlinear Fletcher-Reeves method may be able to achieve the same quadratic convergence within n iterations as the linear Fletcher-Reeves method, however for nonlinear objective functions the convergence rate is naturally worse or even much worse [6]. Figure 6-4c illustrates the nonlinear Fletcher-Reeves method.



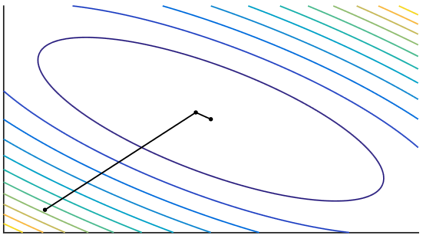
(a) GD method (157 iterations)



(b) NGD method (51 iterations)

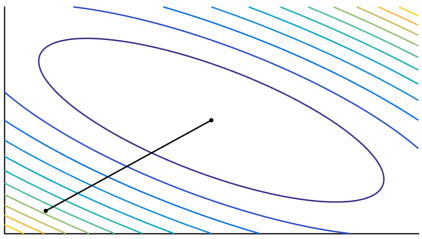


(c) NFR method (37 iterations)

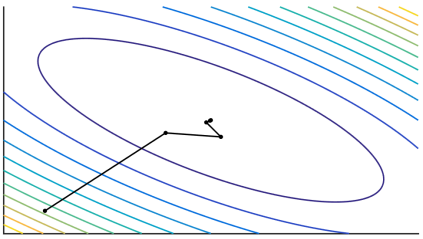


(d) LFR method (2 iterations)

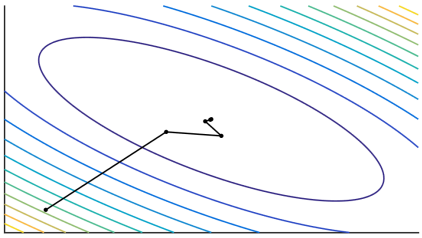
Figure 6-4: First-order optimization of a two-dimensional quadratic function using the gradient descent (GD) method, Nesterov's accelerated gradient descent (NGD) method, nonlinear Fletcher-Reeves (NFR) method and linear Fletcher-Reeves (LFR) method. GD, NGD and NFR use a fixed step size, LFR uses the optimal step size.



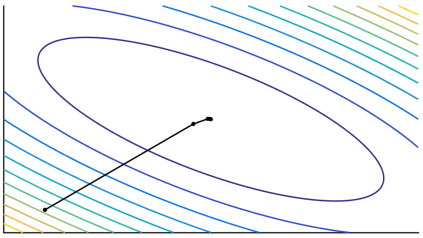
(a) N method (1 iteration)



(b) DFP method (8 iterations)



(c) BFGS method (8 iterations)



(d) LM method (6 iterations)

Figure 6-5: Second-order optimization of a two-dimensional quadratic function using the Newton (N) method, Davidon-Fletcher-Powell (DFP) method, Broyden-Fletcher-Goldfarb-Shanno (BFGS) method and Levenberg-Marquardt (LM) method. DFP and BFGS use a fixed step size, N and LM use the unit step size.

6-3-2 Second-Order Methods

First-order methods may require many iterations to reach the optimum. This is due to the fact that the gradient does not necessarily point towards the optimum, but merely points out the steepest descent direction. In order to converge to the optimum faster, higher-order derivatives can be used, such as the Hessian, which is denoted by $\boldsymbol{H}_k = \nabla^2_{\boldsymbol{x}} \hat{f}_i(\boldsymbol{x}_k)$.

Newton method

The Newton (N) method uses the search direction

$$\boldsymbol{p}_k = -\boldsymbol{H}_k^{-1} \boldsymbol{r}_k. \tag{6-19}$$

If α_k is chosen properly, the Newton method converges to the nearest local optimum within a finite amount of iterations, moreover it does so with quadratic convergence rate [6]. For quadratic functions with a symmetric and positive-definite Hessian, by choosing $\alpha_k = 1$, the Newton method converges to the global optimum within a single iteration. Figure 6-5a illustrates the Newton method.

The Newton method is a very effective method, but it requires the Hessian to be available and nonsingular, and for some functions this may be problematic. Quasi-Newton methods try to resolve this problem by either approximating the Hessian or approximating the inverse of the Hessian [6].

Davidon-Fletcher-Powell method

The Davidon-Fletcher-Powell (DFP) method is a quasi-Newton method which approximates the inverse of the Hessian iteratively. Starting off with a certain initial inverse Hessian approximation the search direction

$$\boldsymbol{p}_k = -\boldsymbol{B}_k \boldsymbol{r}_k \tag{6-20}$$

is used, after which the next inverse Hessian approximation can be computed using

$$\boldsymbol{B}_{k+1} = \boldsymbol{B}_k - \frac{\boldsymbol{B}_k \boldsymbol{\theta}_k \boldsymbol{\theta}_k^T \boldsymbol{B}_k}{\boldsymbol{\theta}_k^T \boldsymbol{B}_k \boldsymbol{\theta}_k} + \frac{\boldsymbol{\kappa}_k \boldsymbol{\kappa}_k^T}{\boldsymbol{\kappa}_k^T \boldsymbol{\kappa}_k}, \tag{6-21}$$

where the vectors

$$\boldsymbol{\theta}_k = \boldsymbol{x}_{k+1} - \boldsymbol{x}_k, \tag{6-22}$$

$$\boldsymbol{\kappa}_k = \boldsymbol{r}_{k+1} - \boldsymbol{r}_k, \tag{6-23}$$

store information about the previous and current location and gradient. For large values of k this results in $\boldsymbol{B}_k \approx \boldsymbol{H}_k^{-1}$. If α_k is chosen such that the Wolfe conditions are satisfied [6],

the DFP method converges to the nearest local optimum within a finite amount of iterations, moreover it does so with a superlinear convergence rate [6]. The DFP method requires more iterations than the Newton method, but much less iterations than the gradient descent method, without requiring computation and inversion of the actual Hessian. Figure 6-5b illustrates the DFP method.

Broyden-Fletcher-Goldfarb-Shanno method

The Broyden-Fletcher-Goldfarb-Shanno (BFGS) method is a quasi-Newton method which approximates the inverse of the Hessian iteratively. The BFGS method is an improved version of the DFP method [6]. Starting off with a certain initial inverse Hessian approximation, the same search direction as for the DFP method (6-20) is used, after which the next inverse Hessian approximation can be computed using

$$\boldsymbol{B}_{k+1} = \left(\boldsymbol{I}_n - \frac{\boldsymbol{\kappa}_k \boldsymbol{\theta}_k^T}{\boldsymbol{\kappa}_k^T \boldsymbol{\theta}_k}\right)^T \boldsymbol{B}_k \left(\boldsymbol{I}_n - \frac{\boldsymbol{\kappa}_k \boldsymbol{\theta}_k^T}{\boldsymbol{\kappa}_k^T \boldsymbol{\theta}_k}\right) + \frac{\boldsymbol{\theta}_k \boldsymbol{\theta}_k^T}{\boldsymbol{\kappa}_k^T \boldsymbol{\theta}_k} = \boldsymbol{V}_k^T \boldsymbol{B}_k \boldsymbol{V}_k + \frac{\boldsymbol{\theta}_k \boldsymbol{\theta}_k^T}{\boldsymbol{\kappa}_k^T \boldsymbol{\theta}_k}. \tag{6-24}$$

For large values of k this results in $\boldsymbol{B}_k \approx \boldsymbol{H}_k^{-1}$. If α_k is chosen such that it satisfies the Wolfe conditions [6], the BFGS method converges to the nearest local optimum within a finite amount of iterations, moreover it does so with a superlinear convergence rate [6]. Similar to the DFP method, the BFGS method requires more iterations than the Newton method, but much less iterations than the gradient descent method, without requiring computation and inversion of the actual Hessian. Figure 6-5c illustrates the BFGS method.

Limited memory Broyden-Fletcher-Goldfarb-Shanno method

For optimization problems with large input dimensions, the inverse Hessian approximation takes the form of a very large and dense matrix which can take up a lot of memory storage. In this case the limited memory Broyden-Fletcher-Goldfarb-Shanno (LBFGS) method can be used [36]. Instead of storing $\boldsymbol{B}_k$, a set of vectors $\{\boldsymbol{\theta}_j \in \mathbb{R}^n, \boldsymbol{\kappa}_j \in \mathbb{R}^n\}_{j=1\ldots u}$ is stored. Every iteration, new values are calculated for $\boldsymbol{\theta}_k$ and $\boldsymbol{\kappa}_k$, which replace the oldest of their respective values already stored. The inverse Hessian approximation can be computed using the stored vectors. This method yields satisfactory results for $3 \leq u \leq 20$ [6]. Since the BFGS method requires storage of $\boldsymbol{B}_k \in \mathbb{R}^{n \times n}$, $\boldsymbol{\theta}_k \in \mathbb{R}^n$ and $\boldsymbol{\kappa}_k \in \mathbb{R}^n$ for a single value of k and the LBFGS method requires storage of $\boldsymbol{\theta}_j \in \mathbb{R}^n$ and $\boldsymbol{\kappa}_j \in \mathbb{R}^n$ for u values of j, only for

$$\begin{aligned} n^2 + 2n &\gg 2un, \\ \frac{1}{2}n + 1 &\gg u, \end{aligned}$$

the LBFGS method will drastically reduce the amount of required memory storage. Moreover it has the same convergence properties as the BFGS method.

Levenberg-Marquardt method

The Levenberg-Marquardt (LM) method approximates the Hessian using

$$\boldsymbol{B}_k^{-1} = \tau \boldsymbol{I}_n + \boldsymbol{H}_k, \tag{6-25}$$

where $\tau \in \mathbb{R}$ is a factor which can prevent $\boldsymbol{B}_k^{-1} \in \mathbb{R}^{n \times n}$ from becoming singular. The same search direction as for the DFP and BFGS methods is used (6-20). Note that for very small values of τ the Levenberg-Marquardt method mimics the Newton method, whereas for very large values of τ it mimics the gradient descent method. If α_k is chosen properly, the Levenberg-Marquardt method converges to the nearest local optimum within a finite amount of iterations, moreover its convergence rate will be somewhere between superlinear and quadratic, depending on the value of τ [6]. Figure 6-5d illustrates the Levenberg-Marquardt method.

6-4 Handling Constraints

The objective function f is constrained by a feasible region $\mathcal{X}$, which is defined by a lower bound $\boldsymbol{x}_{lb} \in \mathbb{R}^n$ and an upper bound $\boldsymbol{x}_{ub} \in \mathbb{R}^n$. This same feasible region also applies to the surrogate function $\hat{f}_i$, hence the optimization methods need to be able to deal with these constraints. Fortunately the constraints are not very complicated, the feasible region $\mathcal{X}$ can be considered a hyper-rectangle. This allows the optimization method to handle the constraints simply by projecting its trajectory onto the feasible region, using

$$\boldsymbol{x}_{k+1} = \min(\max(\boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k, \boldsymbol{x}_{lb}), \boldsymbol{x}_{ub}) \tag{6-26}$$

where min() and max() are element-wise operators which respectively return the smallest and the largest value of their arguments.

6-5 Stopping Conditions

In order to determine when to stop iterating, several stopping conditions need to be established. There are several reasons why an optimization method should be terminated:

- An optimum is reached.
- No more improvement can be achieved without leaving the feasible region.
- The trajectory does not seem to be converging.

Reached optimum

A characteristic of an optimum is that the gradient is zero at that location. Hence it makes sense to stop iterating once

$$\frac{1}{\sqrt{n}} \left|\left| \nabla_{\boldsymbol{x}} \hat{f}_i(\boldsymbol{x}_k) \right|\right|_2 \leq \varepsilon \tag{6-27}$$

is satisfied, where $\varepsilon > 0$ is a marginally low value. The reason for using the RMS value of the gradient in this condition rather than just the norm is that the RMS value ensures equal comparison for all values of n. Note that this condition may be satisfied for maxima as well as for minima.

Edge of the feasible region

There are several methods for dealing with constraints. For rather complicated constraints the Karush-Kuhn-Tucker conditions are a very clean way of checking which constraints are violated and if any improvement within the feasible region is possible [6]. However in the case of the DONE algorithm the feasible region is a hyper-rectangle, which allows the condition

$$\frac{1}{\sqrt{n}} \left|\left| \boldsymbol{x}_{k+1} - \boldsymbol{x}_k \right|\right|_2 \leq \varepsilon \tag{6-28}$$

to terminate the optimization procedure when it is stuck at the edge of the feasible region. Basically this indicates that there is no significant difference between the previous point and the current point, which means that the optimization is stuck at that point.

Divergence

If the optimization procedure does not seem to be converging, it should be terminated as well. The easiest way to do this is by setting a maximum amount of iterations which should not be exceeded.

6-6 Analytical Method

Since analytical expressions for the gradient (3-12) and Hessian (3-13) of $\hat{f}_i$ are available, the solution to the optimization problem (6-1,6-2) may also be found using analytical optimization. Basic calculus implies that the optimum is located where

$$\nabla_{\boldsymbol{x}} \hat{f}_i(\boldsymbol{x}_i) = 0, \tag{6-29}$$

$$\nabla^2_{\boldsymbol{x}} \hat{f}_i(\boldsymbol{x}_i) > 0, \tag{6-30}$$

provided that $\boldsymbol{x}_i \in \mathcal{X}$. Since the gradient is denoted by

$$\nabla_{\boldsymbol{x}} \hat{f}_i(\boldsymbol{x}) = -\boldsymbol{\Omega}\,\mathrm{diag}(\boldsymbol{w}_i)\sin(\boldsymbol{\Omega}^T\boldsymbol{x} + \boldsymbol{b}), \tag{6-31}$$

and the term $-\boldsymbol{\Omega}\,\mathrm{diag}(\boldsymbol{w}_i)$ does not contain the vector $\boldsymbol{x}$, the optimum may be found by imposing

$$\sin(\boldsymbol{\Omega}^T\boldsymbol{x} + \boldsymbol{b}) = \boldsymbol{z}, \tag{6-32}$$
$$\boldsymbol{z} \in \ker(-\boldsymbol{\Omega}\,\mathrm{diag}(\boldsymbol{w}_i)), \tag{6-33}$$

which leads to

$$\boldsymbol{x}_i = (\boldsymbol{\Omega}\boldsymbol{\Omega}^T)^{-1}\boldsymbol{\Omega}(\arcsin(\boldsymbol{z}) - \boldsymbol{b}), \tag{6-34}$$

where the second condition (6-30) needs to be satisfied in order for the found optimum to be a minimum instead of a maximum or a saddle point. Moreover a translation of the optimum may be required to ensure $\boldsymbol{x}_i \in \mathcal{X}$. Obviously this method does not require an iterative scheme, as do the numerical optimization methods, however it does require a kernel computation and solving a set of linear equations. Moreover it should be kept in mind that the arcsin() operator does not return finite values for all possible arguments and that the vector $\boldsymbol{z}$ is not unique. Further research is required to investigate if this approach leads to a practical method for finding $\boldsymbol{x}_i$.

6-7 Comparison

Since the analytical method discussed in Section 6-6 requires the research to diverge from the other methods, it is not taken into account here. Furthermore since the surrogate function $\hat{f}_i$ is a nonlinear function, the LFR method cannot be used in this case. Table 6-1 lists the ranking of all discussed optimization methods based on the parallel computing criteria established in Section 4-7. Here the primary computations are the computations that are inherent to the methods and the secondary computations are the computations that are required to compute gradient (3-12), Hessian (3-13) and function (3-11) values.

Computational complexity

All first-order methods have primary computational complexity $\mathcal{O}(n)$, however due to the computation of the gradient (3-12) they have secondary computational complexity $\mathcal{O}(mn)$. Similarly the second-order methods have primary computational complexity $\mathcal{O}(n^2)$ and secondary computational complexity $\mathcal{O}(mn)$, with the exception of the N and LM methods, which have primary computational complexity $\mathcal{O}(n^3)$ due to the fact that they require a matrix inversion and secondary computational complexity $\mathcal{O}(mn^2)$ due to the computation of the Hessian (3-13).

Method	Primary	Secondary
GD	$\mathcal{O}(n)$	$\mathcal{O}(mn)$
NGD	$\mathcal{O}(n)$	$\mathcal{O}(mn)$
NFR	$\mathcal{O}(n)$	$\mathcal{O}(mn)$
N	$\mathcal{O}(n^3)$	$\mathcal{O}(mn^2)$
DFP	$\mathcal{O}(n^2)$	$\mathcal{O}(mn)$
BFGS	$\mathcal{O}(n^2)$	$\mathcal{O}(mn)$
LBFGS	$\mathcal{O}(n^2)$	$\mathcal{O}(mn)$
LM	$\mathcal{O}(n^3)$	$\mathcal{O}(mn^2)$

(a) Computational complexity

Method	Primary	Secondary
GD	+++++	+++
NGD	+++++	+++
NFR	+++	+++
N	+	+++
DFP	+++	+++
BFGS	+++	+++
LBFGS	+++	+++
LM	+	+++

(b) Degree of parallelism

Method	Primary	Secondary
GD	$\mathcal{O}(n)$	$\mathcal{O}(mn)$
NGD	$\mathcal{O}(n)$	$\mathcal{O}(mn)$
NFR	$\mathcal{O}(n)$	$\mathcal{O}(mn)$
N	$\mathcal{O}(n^2)$	$\mathcal{O}(mn)$
DFP	$\mathcal{O}(n^2)$	$\mathcal{O}(mn)$
BFGS	$\mathcal{O}(n^2)$	$\mathcal{O}(mn)$
LBFGS	$\mathcal{O}(n)$	$\mathcal{O}(mn)$
LM	$\mathcal{O}(n^2)$	$\mathcal{O}(mn)$

(c) Required memory storage

Method	Suitability
GD	Yes
NGD	Yes
NFR	Yes
N	No
DFP	No
BFGS	Yes
LBFGS	No
LM	No

(d) Overall suitability

Table 6-1: Optimization methods judged based on parallel computing criteria. The primary columns indicate the inherent properties of the methods, the secondary columns indicate the properties that are imposed by gradient calculations, Hessian calculations and function evaluations.

Degree of parallelism

Since the GD and NGD methods only require pure vector computations, primarily they are fully parallelizable. NFR, DFP, BFGS and LBFGS all require operations in the form of (4-4), hence their degree of parallelism is medium. Since the N and LM methods require a matrix inversion, they have a very low degree of parallelism. Since the gradient (3-12) and Hessian (3-13) computation require operations in the form of (4-4), all methods have a medium secondary degree of parallelism.

Memory storage

Since the first-order methods only need to store vectors and no matrices, they primarily only require $\mathcal{O}(n)$ memory storage. Due to its special structure, this also applies to the second-order method LBFGS. All other methods primarily require $\mathcal{O}(n^2)$ memory storage. Since for all methods $\mathbf{\Omega} \in \mathbb{R}^{n \times m}$ needs to be stored anyway, secondarily $\mathcal{O}(mn)$ memory storage is required.

Overall suitability

The N and LM methods are dismissed based on their high primary and secondary computational complexity and their low degree of parallelism. The DFP method is dismissed solely because it is stated that BFGS is the improved version of DFP [6]. Based on the memory storage LBFGS should be chosen over BFGS, since they score the same on all other criteria. However, implementing LBFGS is expected to be much more complicated than implementing BFGS since the update relations for LBFGS are much more complicated [6]. Moreover, since generally $m > n$, and all methods secondarily require $\mathcal{O}(mn)$ memory storage, having to store a matrix with memory storage $\mathcal{O}(n^2)$ is not the limiting factor. Hence BFGS is selected and LBFGS is dismissed.

Chapter 7

Implementation

As discussed in Chapters 5 and 6, several methods are available to perform the RFE fitting step and the optimization step of the DONE algorithm respectively. They all have their own characteristics, indicating how suitable they are for large-scale problems and for parallel computing. This chapter discusses how these methods can be implemented, both sequentially and in parallel.

General implementation properties

As discussed in Section 4-2 all parallel implementations use single-precision floating-points. In order to acquire a fair comparison, all sequential implementations use single-precision floating-points as well. The only exception to this rule is the existing sequential DONE implementation, which uses double-precision floating-points.

Furthermore it is always wise not to store more data than absolutely necessary. Triangular, symmetric and diagonal matrices do not need to be fully stored. For instance the symmetric matrix $\boldsymbol{P} \in \mathbb{R}^{m \times m}$ only requires $\frac{1}{2}(m^2 + m)$ floating-points to be stored.

7-1 RFE Fitting

The RFE fitting step can either be performed using optimization-based methods or using recursive methods. The optimization-based methods that are to be implemented are gradient descent (GD), Nesterov's accelerated gradient descent (NGD), nonlinear Fletcher-Reeves (NFR) and linear Fletcher-Reeves (LFR). The recursive methods that are to be implemented are recursive least-squares (RLS) and inverse QR (IQR). GD is only included as basis for the NGD, NFR and LFR methods.

7-1-1 Characteristic Operations

All RFE fitting implementations can be divided into several blocks of operations that are characteristic for the RFE fitting framework. These operations are present in all or most of the RFE fitting implementations.

Initial values

The initial value $\boldsymbol{w}_0 = \mathbf{0}_m$ is used for all RFE fitting implementations. The initial values $\boldsymbol{\Phi}_0 = \lambda \boldsymbol{I}_m$ and $\boldsymbol{s}_0 = \mathbf{0}_m$ are used in all optimization-based RFE fitting implementations. The RLS implementation uses the initial value $\boldsymbol{P}_0 = \lambda^{-1}\boldsymbol{I}_m$ and the IQR implementation uses the initial value $\boldsymbol{P}_0^{\frac{1}{2}} = \lambda^{-\frac{1}{2}}\boldsymbol{I}_m$, where $\boldsymbol{P}_i = \boldsymbol{\Phi}_i^{-1}$.

Inner and outer loop

All RFE fitting implementations are based on an outer loop, which takes consecutive measurements and passes these measurements to the RFE fitting method at hand to be processed. The optimization-based RFE fitting implementations require an inner loop, which reflects their iterative nature. The counter $i \in \mathbb{Z}$ counts the measurements (outer loop), whereas the counter $k \in \mathbb{Z}$ counts the iterations (inner loop). The measurement counter has a fixed maximum value $N \in \mathbb{Z}$ whereas the iteration counter k does not have a fixed maximum value. Note that the parameter vector $\boldsymbol{w}_k$ is not reset to $\boldsymbol{w}_k = \mathbf{0}_m$ every time the outer loop is repeated. This ensures that the new surrogate function is built upon the old surrogate function instead of computing a completely new one from scratch, thus reducing the amount of required iterations by using the information that is already available.

Taking measurements

RFE fitting can be applied to any function $f : \mathbb{R}^n \mapsto \mathbb{R}$. The point $\boldsymbol{x}_i$ can either be drawn from a certain probability distribution, or be determined using clever estimation techniques, as does the DONE algorithm.

Domain conversion

All measurements are taken in the input domain $\mathcal{X} \subset \mathbb{R}^n$, however all data are stored in the RFE domain $\mathcal{A} \subseteq \mathbb{R}^m$. This means that measurements need to be converted from the former to the latter, using the relation

$$\boldsymbol{a}_i = \cos(\boldsymbol{\Omega}^T \boldsymbol{x}_i + \boldsymbol{b}), \tag{7-1}$$

where $\boldsymbol{x}_i \in \mathcal{X}$ and $\boldsymbol{a}_i \in \mathcal{A}$.

Update relations

Every time a new measurement is available, some updates can be performed. For the optimization-based implementations, the least-squares cost function (5-2) is updated by updating $\boldsymbol{\Phi}_i$ and $\boldsymbol{s}_i$ using the update relations (5-7, 5-8). For the RLS implementation, the new value $\boldsymbol{w}_i$ can be immediately computed using the update relations (5-9, 5-10, 5-11, 5-12). For the IQR implementation, this can be done as well, using the IQR update relation (5-14) combined with the final RLS update relation (5-12).

Gradient calculation

For all optimization-based implementations, calculating the gradient (5-4) is required. For the sake of simplicity, the least-squares cost function (5-3) can be transformed to

$$\hat{J}_i(\boldsymbol{w}) = \frac{1}{2} J_i(\boldsymbol{w}) - \boldsymbol{y}_i^T \boldsymbol{y}_i = \frac{1}{2} \boldsymbol{w}^T \boldsymbol{\Phi}_i \boldsymbol{w} - \boldsymbol{w}^T \boldsymbol{s}_i, \tag{7-2}$$

without changing the location of the optimum. This introduces a slightly simpler representation of the gradient in the form of

$$\boldsymbol{r}_k = \boldsymbol{\Phi}_i \boldsymbol{w}_k - \boldsymbol{s}_i. \tag{7-3}$$

Step size selection

All optimization-based implementations require a step size to be selected. For the NGD implementation, a requirement is that the step size α_i is a Lipschitz constant of the gradient (7-3) and hence represents the largest possible value of the Hessian $\boldsymbol{\Phi}_i$ [31, 33]. In order to make sure that this requirement is satisfied, the step size

$$\alpha_i = \frac{1}{10} \, ||\boldsymbol{\Phi}_i||_F^{-1} \tag{7-4}$$

is used. Here the Frobenius norm is used to determine a scalar value from a matrix and the factor $\frac{1}{10}$ acts as a safety margin. Since this step size should lead to steps that are not too large and not too small either, it is used in the GD and NFR implementations as well. The LFR implementation has its own way of computing the step size, such that it is optimal for every iteration.

Search direction

All optimization-based implementations require a search direction $\boldsymbol{p}_k$. The GD implementation uses the negative gradient as search direction, the NGD implementation has no explicit declaration of the search direction but also incorporates the gradient into its choice of trajectory and the NFR and LFR implementations start off with the negative gradient and gradually improve the search direction each iteration.

Trajectory

Moving along the trajectory can be performed by taking a step along the selected search direction with the selected step size. This holds for all optimization-based implementations, except for the NGD implementation. The NGD implementation needs to determine a trajectory for the momentum vector $\boldsymbol{v}_k$ as well as for the weight vector $\boldsymbol{w}_k$. The value for the momentum vector is chosen to be $\boldsymbol{v}_k = \boldsymbol{x}_k$ every time the outer loop is repeated.

Convergence test

The inner loop that is present in all optimization-based implementations continues as long as the RMS value of the gradient is larger than a certain predefined error margin ε. The RMS value can be obtained by computing the two-norm and dividing it by the square root of the dimension m, as in (6-27).

Although the LFR implementation is guaranteed to convergence within m iterations, by maintaining ε as stopping condition rather than blindly performing m iterations, convergence may even occur in less than m iterations.

7-1-2 Mathematical Implementation

The RFE fitting implementations are represented by Algorithms 1 to 6. Table 7-1 lists all characteristic operations and at which lines they are incorporated.

Characteristic operations	**GD**	**NGD**	**NFR**	**LFR**	**RLS**	**IQR**
Initial values	1 - 3	1 - 3	1 - 3	1 - 3	1, 2	1, 2
Outer loop	5 - 22	5 - 26	5 - 24	5 - 24	3 - 14	3 - 12
Inner loop	14 - 19	16 - 23	15 - 21	14 - 21		
Taking measurements	6, 7	6, 7	6, 7	6, 7	4, 5	4, 5
Domain conversion	9	9	9	9	7	7
Update relations	10, 11	10, 11	10, 11	10, 11	8 - 11	8, 9
Gradient calculation	12, 17	12, 21	12, 17	12, 17		
Step size selection	13	14	14	15		
Search direction	15		13, 19	13, 19		
Trajectory	16	13, 19, 20	16	16		
Convergence test	14	16	15	14		

Table 7-1: Characteristic operations for the RFE fitting step. The numbers in this table refer to line numbers in Algorithms 1 to 6, which represent the GD, NGD, NFR, LFR, RLS and IQR implementations respectively.

Algorithm 1 RFE fitting using the GD method

$\boldsymbol{w}_0 \leftarrow \mathbf{0}_m$
$\boldsymbol{\Phi}_0 \leftarrow \lambda \boldsymbol{I}_m$
$\boldsymbol{s}_0 \leftarrow \mathbf{0}_m$
$k \leftarrow 0$
for $i = 1 : N$ **do**
  $\boldsymbol{x}_i \leftarrow$ input measurement
  $y_i \leftarrow$ output measurement
  procedure RFEGD($\boldsymbol{x}_i$, y_i, $\boldsymbol{\Phi}_{i-1}$, $\boldsymbol{s}_{i-1}$, $\boldsymbol{w}_k$, $\boldsymbol{\Omega}$, $\boldsymbol{b}$, ε)
    $\boldsymbol{a}_i \leftarrow \cos(\boldsymbol{\Omega}^T \boldsymbol{x}_i + \boldsymbol{b})$
    $\boldsymbol{\Phi}_i \leftarrow \boldsymbol{\Phi}_{i-1} + \boldsymbol{a}_i \boldsymbol{a}_i^T$
    $\boldsymbol{s}_i \leftarrow \boldsymbol{s}_{i-1} + y_i \boldsymbol{a}_i$
    $\boldsymbol{r}_k \leftarrow \boldsymbol{\Phi}_i \boldsymbol{w}_k - \boldsymbol{s}_i$
    $\alpha_i \leftarrow \frac{1}{10} ||\boldsymbol{\Phi}_i||_F^{-1}$
    while $||\boldsymbol{r}_k||_2 \, m^{-\frac{1}{2}} > \varepsilon$ **do**
      $\boldsymbol{p}_k \leftarrow -\boldsymbol{r}_k$
      $\boldsymbol{w}_{k+1} \leftarrow \boldsymbol{w}_k + \alpha_i \boldsymbol{p}_k$
      $\boldsymbol{r}_{k+1} \leftarrow \boldsymbol{\Phi}_i \boldsymbol{w}_{k+1} - \boldsymbol{s}_i$
      $k \leftarrow k + 1$
    end while
    return $\boldsymbol{\Phi}_i$, $\boldsymbol{s}_i$, $\boldsymbol{w}_k$
  end procedure
end for

Algorithm 2 RFE fitting using the NGD method

$\boldsymbol{w}_0 \leftarrow \mathbf{0}_m$
$\boldsymbol{\Phi}_0 \leftarrow \lambda \boldsymbol{I}_m$
$\boldsymbol{s}_0 \leftarrow \mathbf{0}_m$
$k \leftarrow 0$
for $i = 1 : N$ **do**
  $\boldsymbol{x}_i \leftarrow$ input measurement
  $y_i \leftarrow$ output measurement
  procedure RFENGD($\boldsymbol{x}_i$, y_i, $\boldsymbol{\Phi}_{i-1}$, $\boldsymbol{s}_{i-1}$, $\boldsymbol{w}_k$, $\boldsymbol{\Omega}$, $\boldsymbol{b}$, ε)
    $\boldsymbol{a}_i \leftarrow \cos(\boldsymbol{\Omega}^T \boldsymbol{x}_i + \boldsymbol{b})$
    $\boldsymbol{\Phi}_i \leftarrow \boldsymbol{\Phi}_{i-1} + \boldsymbol{a}_i \boldsymbol{a}_i^T$
    $\boldsymbol{s}_i \leftarrow \boldsymbol{s}_{i-1} + y_i \boldsymbol{a}_i$
    $\boldsymbol{r}_k \leftarrow \boldsymbol{\Phi}_i \boldsymbol{w}_k - \boldsymbol{s}_i$
    $\boldsymbol{v}_k \leftarrow \boldsymbol{w}_k$
    $\alpha_i \leftarrow \frac{1}{10} ||\boldsymbol{\Phi}_i||_F^{-1}$
    $\tau_k \leftarrow 0$
    while $||\boldsymbol{r}_k||_2 \, m^{-\frac{1}{2}} > \varepsilon$ **do**
      $\tau_{k+1} \leftarrow \frac{1}{2} + \frac{1}{2}\sqrt{1 + 4\tau_k^2}$
      $\gamma_k \leftarrow \tau_{k+1}(1 - \tau_k)^{-1}$
      $\boldsymbol{v}_{k+1} \leftarrow \boldsymbol{w}_k - \alpha_i \boldsymbol{r}_k$
      $\boldsymbol{w}_{k+1} \leftarrow (1 - \gamma_k)\boldsymbol{v}_{k+1} + \gamma_k \boldsymbol{v}_k$
      $\boldsymbol{r}_{k+1} \leftarrow \boldsymbol{\Phi}_i \boldsymbol{w}_{k+1} - \boldsymbol{s}_i$
      $k \leftarrow k + 1$
    end while
    return $\boldsymbol{\Phi}_i$, $\boldsymbol{s}_i$, $\boldsymbol{w}_k$
  end procedure
end for

Algorithm 3 RFE fitting using the NFR method

$\boldsymbol{w}_0 \leftarrow \mathbf{0}_m$
$\boldsymbol{\Phi}_0 \leftarrow \lambda \boldsymbol{I}_m$
$\boldsymbol{s}_0 \leftarrow \mathbf{0}_m$
$k \leftarrow 0$
for $i = 1 : N$ **do**
  $\boldsymbol{x}_i \leftarrow$ input measurement
  $y_i \leftarrow$ output measurement
  procedure RFENFR($\boldsymbol{x}_i$, y_i, $\boldsymbol{\Phi}_{i-1}$, $\boldsymbol{s}_{i-1}$, $\boldsymbol{w}_k$, $\boldsymbol{\Omega}$, $\boldsymbol{b}$, ε)
    $\boldsymbol{a}_i \leftarrow \cos(\boldsymbol{\Omega}^T \boldsymbol{x}_i + \boldsymbol{b})$
    $\boldsymbol{\Phi}_i \leftarrow \boldsymbol{\Phi}_{i-1} + \boldsymbol{a}_i \boldsymbol{a}_i^T$
    $\boldsymbol{s}_i \leftarrow \boldsymbol{s}_{i-1} + y_i \boldsymbol{a}_i$
    $\boldsymbol{r}_k \leftarrow \boldsymbol{\Phi}_i \boldsymbol{w}_k - \boldsymbol{s}_i$
    $\boldsymbol{p}_k \leftarrow -\boldsymbol{r}_k$
    $\alpha_i \leftarrow \frac{1}{10} ||\boldsymbol{\Phi}_i||_F^{-1}$
    while $||\boldsymbol{r}_k||_2 \, m^{-\frac{1}{2}} > \varepsilon$ **do**
      $\boldsymbol{w}_{k+1} \leftarrow \boldsymbol{w}_k + \alpha_i \boldsymbol{p}_k$
      $\boldsymbol{r}_{k+1} \leftarrow \boldsymbol{\Phi}_i \boldsymbol{w}_{k+1} - \boldsymbol{s}_i$
      $\tau_k \leftarrow (\boldsymbol{r}_{k+1}^T \boldsymbol{r}_{k+1})(\boldsymbol{r}_k^T \boldsymbol{r}_k)^{-1}$
      $\boldsymbol{p}_{k+1} \leftarrow -\boldsymbol{r}_{k+1} + \tau_k \boldsymbol{p}_k$
      $k \leftarrow k + 1$
    end while
    return $\boldsymbol{\Phi}_i$, $\boldsymbol{s}_i$, $\boldsymbol{w}_k$
  end procedure
end for

Algorithm 4 RFE fitting using the LFR method

$\boldsymbol{w}_0 \leftarrow \mathbf{0}_m$
$\boldsymbol{\Phi}_0 \leftarrow \lambda \boldsymbol{I}_m$
$\boldsymbol{s}_0 \leftarrow \mathbf{0}_m$
$k \leftarrow 0$
for $i = 1 : N$ **do**
  $\boldsymbol{x}_i \leftarrow$ input measurement
  $y_i \leftarrow$ output measurement
  procedure RFELFR($\boldsymbol{x}_i$, y_i, $\boldsymbol{\Phi}_{i-1}$, $\boldsymbol{s}_{i-1}$, $\boldsymbol{w}_k$, $\boldsymbol{\Omega}$, $\boldsymbol{b}$, ε)
    $\boldsymbol{a}_i \leftarrow \cos(\boldsymbol{\Omega}^T \boldsymbol{x}_i + \boldsymbol{b})$
    $\boldsymbol{\Phi}_i \leftarrow \boldsymbol{\Phi}_{i-1} + \boldsymbol{a}_i \boldsymbol{a}_i^T$
    $\boldsymbol{s}_i \leftarrow \boldsymbol{s}_{i-1} + y_i \boldsymbol{a}_i$
    $\boldsymbol{r}_k \leftarrow \boldsymbol{\Phi}_i \boldsymbol{w}_k - \boldsymbol{s}_i$
    $\boldsymbol{p}_k \leftarrow -\boldsymbol{r}_k$
    while $||\boldsymbol{r}_k||_2 \, m^{-\frac{1}{2}} > \varepsilon$ **do**
      $\alpha_k \leftarrow (\boldsymbol{r}_k^T \boldsymbol{r}_k)(\boldsymbol{p}_k^T \boldsymbol{\Phi}_i \boldsymbol{p}_k)^{-1}$
      $\boldsymbol{w}_{k+1} \leftarrow \boldsymbol{w}_k + \alpha_k \boldsymbol{p}_k$
      $\boldsymbol{r}_{k+1} \leftarrow \boldsymbol{\Phi}_i \boldsymbol{w}_{k+1} - \boldsymbol{s}_i$
      $\tau_k \leftarrow (\boldsymbol{r}_{k+1}^T \boldsymbol{r}_{k+1})(\boldsymbol{r}_k^T \boldsymbol{r}_k)^{-1}$
      $\boldsymbol{p}_{k+1} \leftarrow -\boldsymbol{r}_{k+1} + \tau_k \boldsymbol{p}_k$
      $k \leftarrow k + 1$
    end while
    return $\boldsymbol{\Phi}_i$, $\boldsymbol{s}_i$, $\boldsymbol{w}_k$
  end procedure
end for

Algorithm 5 RFE fitting using the RLS method

$\boldsymbol{w}_0 \leftarrow \mathbf{0}_m$
$\boldsymbol{P}_0 \leftarrow \lambda^{-1}\boldsymbol{I}_m$
for $i = 1 : N$ **do**
  $\boldsymbol{x}_i \leftarrow$ input measurement
  $y_i \leftarrow$ output measurement
  procedure RFERLS($\boldsymbol{x}_i$, y_i, $\boldsymbol{P}_{i-1}$, $\boldsymbol{w}_{i-1}$, $\boldsymbol{\Omega}$, $\boldsymbol{b}$)
    $\boldsymbol{a}_i \leftarrow \cos(\boldsymbol{\Omega}^T\boldsymbol{x}_i + \boldsymbol{b})$
    $\gamma_i \leftarrow (1 + \boldsymbol{a}_i^T\boldsymbol{P}_{i-1}\boldsymbol{a}_i)^{-1}$
    $\boldsymbol{g}_i \leftarrow \gamma_i\boldsymbol{P}_{i-1}\boldsymbol{a}_i$
    $\boldsymbol{w}_i \leftarrow \boldsymbol{w}_{i-1} + \boldsymbol{a}_i^T\boldsymbol{w}_{i-1}\boldsymbol{g}_i + y_i\boldsymbol{g}_i$
    $\boldsymbol{P}_i \leftarrow \boldsymbol{P}_{i-1} - \gamma_i^{-1}\boldsymbol{g}_i\boldsymbol{g}_i^T$
    return $\boldsymbol{P}_i$, $\boldsymbol{w}_i$
  end procedure
end for

Algorithm 6 RFE fitting using the IQR method

$\boldsymbol{w}_0 \leftarrow \mathbf{0}_m$
$\boldsymbol{P}_0^{\frac{1}{2}} \leftarrow \lambda^{-\frac{1}{2}}\boldsymbol{I}_m$
for $i = 1 : N$ **do**
  $\boldsymbol{x}_i \leftarrow$ input measurement
  $y_i \leftarrow$ output measurement
  procedure RFEIQR($\boldsymbol{x}_i$, y_i, $\boldsymbol{P}_{i-1}^{\frac{1}{2}}$, $\boldsymbol{w}_{i-1}$, $\boldsymbol{\Omega}$, $\boldsymbol{b}$)
    $\boldsymbol{a}_i \leftarrow \cos(\boldsymbol{\Omega}^T\boldsymbol{x}_i + \boldsymbol{b})$
    $\begin{bmatrix} \gamma_i^{-\frac{1}{2}} & \mathbf{0}_m^T \\ \boldsymbol{g}_i\gamma_i^{-\frac{1}{2}} & \boldsymbol{P}_i^{\frac{1}{2}} \end{bmatrix} \leftarrow \begin{bmatrix} 1 & \boldsymbol{a}_i^T\boldsymbol{P}_{i-1}^{\frac{1}{2}} \\ \mathbf{0}_m & \boldsymbol{P}_{i-1}^{\frac{1}{2}} \end{bmatrix} \boldsymbol{\Theta}$
    $\boldsymbol{w}_i \leftarrow \boldsymbol{w}_{i-1} + \boldsymbol{a}_i^T\boldsymbol{w}_{i-1}\boldsymbol{g}_i + y_i\boldsymbol{g}_i$
    return $\boldsymbol{P}_i^{\frac{1}{2}}$, $\boldsymbol{w}_i$
  end procedure
end for

Algorithm 1 represents RFE fitting using the GD method. This implementation can be obtained using all the discussed characteristic operations and no more. The GD implementation is not expected to be the best performing implementation, neither when looking at the amount of iterations nor when looking at the run time, however it is a good benchmark for the other implementations.

Algorithm 2 represents RFE fitting using the NGD method. The difference between GD and NGD is that instead of only propagating the weight vector $\boldsymbol{w}_k$, the NGD implementation also propagates the momentum vector $\boldsymbol{v}_k$.

Algorithm 3 represents RFE fitting using the NFR method. The difference between GD and NFR is that for the NFR implementation the search direction $\boldsymbol{p}_k$ is improved iteration-wise, ensuring the more efficient convergence rate of conjugate gradient methods.

Algorithm 4 represents RFE fitting using the LFR method. The LFR implementation is exactly the same as the NFR implementation, except for the step size selection. Where for the NFR implementation the step size α_i is chosen to be established every time a new

measurement is available, the LFR implementation establishes the step size α_k every iteration in an optimal way.

Algorithm 5 represents RFE fitting using the RLS method. As listed in Table 7-1, this implementation lacks all the characteristic operations that are used by the optimization-based implementations. The update relations that are used in this case are the basic RLS update relations (5-9, 5-10, 5-11, 5-12).

Algorithm 6 represents RFE fitting using the IQR method. Just like the RLS implementation, the IQR implementation lacks all the characteristic operations that are used by the optimization-based implementations. The update relations that are used are the IQR update relation (5-14) combined with the final RLS update relation (5-12), where the matrix rotation is performed using so-called Givens rotations [14, 29].

7-1-3 Sequential Implementation

Most of the discussed operations can be performed straightforwardly by one of the basic linear algebra subprograms (BLAS) in the CBLAS library. For instance the matrix update at line 10 of Algorithm 1 is a symmetric rank 1 matrix update and the vector update at line 11 is a so-called "axpy" operation.

Not all operations can be performed that straightforwardly however. For instance the vector update at line 10 of Algorithm 5 can only be performed using

$$\boldsymbol{w}_i \leftarrow \boldsymbol{w}_{i-1} + (\boldsymbol{a}_i^T \boldsymbol{w}_{i-1} + y_i)\boldsymbol{g}_i, \tag{7-5}$$

which is a combination of a dot product, a scalar addition and an "axpy" operation. In all RFE fitting implementations there are more of these situations, however this example is sufficient to describe how to deal with these situations.

Besides the operations that require a combination of BLAS and scalar operations, there are also some operations that need to be programmed manually. In all RFE fitting implementations, there are two situations like this.

The first situation is the domain conversion at line 7 of Algorithm 5 and also present in all other RFE fitting implementations. This operation contains a cosine computation, which is not part of the CBLAS library. Hence the domain conversion needs to be performed using

$$\boldsymbol{\delta} \leftarrow \boldsymbol{\Omega}^T \boldsymbol{x}_i + \boldsymbol{b}, \tag{7-6}$$

$$a_j \leftarrow \cos(\delta_j) \quad \forall \quad j \in \{1, \ldots, m\}, \tag{7-7}$$

where the first part (7-6) can be performed using a combination of BLAS and the second part (7-7) has to be performed using a for-loop. Here $\boldsymbol{\delta}$ is a support vector.

The other situation in which manual programming is required, is the matrix rotation at line 8 of Algorithm 6. As discussed this is performed using so-called Givens rotations [14, 29] and although there are BLAS to perform a Givens rotation, these rotations are performed on one column at a time. Hence a for-loop is required to rotate the entire matrix.

7-1-4 Parallel Implementation

For every command in the CBLAS library, there is a parallel equivalent in the CUBLAS library [26]. Therefore it is rather straightforward to turn the sequential implementations into parallel implementations. However the domain conversion needs to be dealt with in a different fashion for parallel computing. The first part (7-6) can be performed using CUBLAS straightforwardly, however for the second part (7-7) the for-loop needs to be replaced by a kernel, performing the computation for each vector element at the same time.

It would be very beneficial if the Givens rotations at line 8 of Algorithm 6 could also be parallelized. And although there are CUBLAS equivalents for the CBLAS commands that perform Givens rotations, the for-loop that performs the Givens rotation column by column cannot be dissolved, since the result of the first column computation is needed to compute the second column, and so on. So despite the fact that $\boldsymbol{P}_i^{\frac{1}{2}}$ is lower-triangular, which diminishes the computational cost of the Givens rotations, this part cannot be made much more effective using parallel computing.

Another important aspect of parallel computing is the data communication. As discussed in Section 4-2 parallel computing is the most effective when using coarse-grained parallelism. Therefore all parallel RFE fitting implementations first initialize $\boldsymbol{\Omega}$, $\boldsymbol{b}$, $\boldsymbol{w}_0$, $\boldsymbol{s}_0$ and $\boldsymbol{\Phi}_0$ (or one of its inverse representations), then copy all data from CPU to GPU, then perform all computations and afterwards only copy $\boldsymbol{w}_k$ back, since that is the only interesting piece of information. Naturally, every time a new measurement is available, $\boldsymbol{x}_i$ needs to be copied from CPU to GPU as well, but this inevitable data copy is the only communication that occurs in between of the computations. Note that copying scalar values is of no significance here so this is not taken into account.

7-2 Optimization

The optimization step can either be performed using first-order methods or using second-order methods. The methods that are to be implemented are the first-order methods gradient descent (GD), Nesterov's accelerated gradient descent (NGD) and nonlinear Fletcher-Reeves (NFR) and the second-order method Broyden-Fletcher-Goldfarb-Shanno (BFGS). Here as well GD is only included as basis for the other methods.

7-2-1 Characteristic Operations

All optimization methods can be divided into several blocks of operations that are characteristic for the optimization framework. These operations are present in all or most of the optimization implementations. Most of the characteristic operations for the optimization implementations are also present in the optimization-based RFE fitting implementations.

Initial values

The initial value $\boldsymbol{x}_0$ is user-supplied for all optimization implementations. The NGD implementation also requires an initial value for the momentum vector, which is always chosen to be $\boldsymbol{v}_0 = \boldsymbol{x}_0$. The BFGS implementation requires an initial value for the inverse Hessian approximation, which is always chosen to be $\boldsymbol{B}_0 = \boldsymbol{I}_0$.

Inner loop

Note that the optimization implementations operate at the same level as the RFE fitting implementations, which is within the outer loop which takes the measurements. Hence the optimization implementations can be considered to be inner loops as well.

Backtracking

Within the discussed inner loop however, lies another nested loop, invoking a backtracking procedure which determines a step size that satisfies the Wolfe conditions (6-6, 6-7) [6]. Where the inner loop counts the iterations, this backtracking loop counts the backtracking iterations. This holds for all optimization implementations, except for the NGD implementation.

Function evaluation

Since the Wolfe conditions are conditions that require function evaluations, the surrogate function (6-2) has to be evaluated several times per iteration. Only for the NGD implementation this is not the case, since it does not involve the Wolfe conditions.

Gradient calculation

For all the optimization implementations, the gradient $\boldsymbol{r}_k$ (3-12) needs to be computed. The gradient value is used for establishing a search direction and for checking the convergence.

Step size selection

For the GD, NFR and BFGS implementations the step size is selected using backtracking. The procedure starts with $\alpha_k = 1$ and than the step size is diminished until the Wolfe conditions are satisfied. For the NGD implementation, the step size α is required to be the inverse of a Lipschitz constant of the gradient (3-12), hence it represents the largest possible value of the Hessian (3-13) [31, 33]. Since for the trigonometric part of the Hessian

$$\text{diag}(\sin(\boldsymbol{\Omega}^T \boldsymbol{x}_k + \boldsymbol{b})) \leq \boldsymbol{I}_m$$

holds, where the $\leq$ operator is an element-wise operator, the largest possible value of the Hessian is denoted by

$$\boldsymbol{H}_{max} = \boldsymbol{\Omega}\,\mathrm{diag}(\boldsymbol{w}_i)\boldsymbol{\Omega}^T.$$

However, since computing this matrix-matrix product has computational complexity $\mathcal{O}(mn^2)$, the rather simpler computation

$$\alpha = \frac{1}{10}\,||\boldsymbol{\Omega}\boldsymbol{w}_i||_F^{-1} \tag{7-8}$$

is performed. Here the Frobenius norm is used to determine a scalar value from a matrix and the factor $\frac{1}{10}$ acts as a safety margin.

Search direction

All optimization implementations require a search direction $\boldsymbol{p}_k$. The GD implementation uses the negative gradient as search direction, the NGD implementation has no explicit declaration of the search direction but also incorporates the gradient into its choice of trajectory, the NFR implementation starts of with the negative gradient and gradually improves the search direction each iteration and the BFGS implementation uses the product of the inverse Hessian approximation and the negative gradient as search direction.

Trajectory

Moving along the trajectory can be performed by taking a step along the selected search direction with the selected step size. This holds for all optimization-based implementations, except for the NGD implementation. The NGD implementation needs to determine a trajectory for the momentum vector $\boldsymbol{v}_k$ as well as for the input vector $\boldsymbol{x}_k$.

Convergence test

The inner loop that is present in all optimization implementations continues as long as the RMS value of the gradient is larger than a certain predefined error margin ε. The RMS value can be obtained by computing the two-norm and dividing it by the square root of the dimension n, as in (6-27).

7-2-2 Mathematical Implementation

The optimization implementations are represented by Algorithms 7 to 10. Table 7-2 lists all characteristic operations and at which lines they are incorporated.

Algorithm 7 represents optimization using the GD method. This implementation can be obtained using all the discussed characteristic operations and no more. The GD implementation is not expected to be the best performing implementation, neither when looking at the amount of iterations nor when looking at the run time, however it is a good benchmark for the other implementations.

Algorithm 7 Optimization using the GD method

procedure OPTIMGD($\boldsymbol{x}_0$, $\boldsymbol{w}$, $\boldsymbol{\Omega}$, $\boldsymbol{b}$, ε)
  $\boldsymbol{r}_0 \leftarrow -\boldsymbol{\Omega}\text{diag}(\boldsymbol{w})\sin(\boldsymbol{\Omega}^T\boldsymbol{x}_0 + \boldsymbol{b})$
  $\gamma_0 \leftarrow \boldsymbol{w}^T\cos(\boldsymbol{\Omega}\boldsymbol{x}_k + \boldsymbol{b})$
  $k \leftarrow 0$
  while $||\boldsymbol{r}_k||_2\, n^{-\frac{1}{2}} > \varepsilon$ **do**
    $\alpha_k \leftarrow 1$
    $\boldsymbol{p}_k \leftarrow -\boldsymbol{r}_k$
    $\boldsymbol{x}_{k+1} \leftarrow \boldsymbol{x}_k + \alpha_k\boldsymbol{p}_k$
    $\gamma_{k+1} \leftarrow \boldsymbol{w}^T\cos(\boldsymbol{\Omega}\boldsymbol{x}_{k+1} + \boldsymbol{b})$
    while $\gamma_{k+1} - \gamma_k - \frac{1}{2}\alpha_k\boldsymbol{r}_k^T\boldsymbol{p}_k > \varepsilon$ **do**
      $\alpha_k \leftarrow \frac{1}{2}\alpha_k$
      $\boldsymbol{x}_{k+1} \leftarrow \boldsymbol{x}_k + \alpha_k\boldsymbol{p}_k$
      $\gamma_{k+1} \leftarrow \boldsymbol{w}^T\cos(\boldsymbol{\Omega}\boldsymbol{x}_{k+1} + \boldsymbol{b})$
    end while
    $\boldsymbol{r}_{k+1} \leftarrow -\boldsymbol{\Omega}\text{diag}(\boldsymbol{w})\sin(\boldsymbol{\Omega}^T\boldsymbol{x}_{k+1} + \boldsymbol{b})$
    $k \leftarrow k + 1$
  end while
  return $\boldsymbol{x}_k$
end procedure

Algorithm 8 Optimization using the NGD method

procedure OPTIMNGD($\boldsymbol{x}_0$, $\boldsymbol{w}$, $\boldsymbol{\Omega}$, $\boldsymbol{b}$, ε)
  $\boldsymbol{r}_0 \leftarrow -\boldsymbol{\Omega}\text{diag}(\boldsymbol{w})\sin(\boldsymbol{\Omega}^T\boldsymbol{x}_0 + \boldsymbol{b})$
  $\boldsymbol{v}_0 \leftarrow \boldsymbol{x}_0$
  $\alpha \leftarrow \frac{1}{10}\,||\boldsymbol{\Omega}\boldsymbol{w}||_F^{-1}$
  $\tau_0 \leftarrow 0$
  $k \leftarrow 0$
  while $||\boldsymbol{r}_k||_2\, n^{-\frac{1}{2}} > \varepsilon$ **do**
    $\tau_{k+1} \leftarrow \frac{1}{2} + \frac{1}{2}\sqrt{1 + 4\tau_k^2}$
    $\gamma_k \leftarrow \tau_{k+1}(1 - \tau_k)^{-1}$
    $\boldsymbol{v}_{k+1} \leftarrow \boldsymbol{x}_k - \alpha\boldsymbol{r}_k$
    $\boldsymbol{x}_{k+1} \leftarrow (1 - \gamma_k)\boldsymbol{v}_{k+1} + \gamma_k\boldsymbol{v}_k$
    $\boldsymbol{r}_{k+1} \leftarrow -\boldsymbol{\Omega}\text{diag}(\boldsymbol{w})\sin(\boldsymbol{\Omega}^T\boldsymbol{x}_{k+1} + \boldsymbol{b})$
    $k \leftarrow k + 1$
  end while
  return $\boldsymbol{x}_k$
end procedure

Algorithm 8 represents optimization using the NGD method. The difference between GD and NGD is that instead of only propagating the input vector $\boldsymbol{x}_k$, the NGD implementation also propagates the momentum vector $\boldsymbol{v}_k$.

Algorithm 9 represents optimization using the NFR method. The difference between GD and NFR is that for the NFR implementation the search direction $\boldsymbol{p}_k$ is improved iteration-wise, ensuring the more efficient convergence rate of conjugate gradient methods.

Algorithm 10 represents optimization using the BFGS method. The difference between GD and BFGS is that BFGS is a second-order method and thus also propagates an approximation for the inverse of the Hessian, $\boldsymbol{B}_k$. Moreover in the BFGS implementation the search direction

is the product of the inverse Hessian approximation and the negative gradient. The inverse Hessian approximation is propagated according to the BFGS update (6-24).

Algorithm 9 Optimization using the NFR method

procedure OPTIMNFR($\boldsymbol{x}_0$, $\boldsymbol{w}$, $\boldsymbol{\Omega}$, $\boldsymbol{b}$, ε)
  $\boldsymbol{r}_0 \leftarrow -\boldsymbol{\Omega}\text{diag}(\boldsymbol{w})\sin(\boldsymbol{\Omega}^T\boldsymbol{x}_0 + \boldsymbol{b})$
  $\gamma_0 \leftarrow \boldsymbol{w}^T\cos(\boldsymbol{\Omega}\boldsymbol{x}_k + \boldsymbol{b})$
  $\boldsymbol{p}_0 \leftarrow -\boldsymbol{r}_0$
  $k \leftarrow 0$
  while $||\boldsymbol{r}_k||_2\, n^{-\frac{1}{2}} > \varepsilon$ **do**
    $\alpha_k \leftarrow 1$
    $\boldsymbol{x}_{k+1} \leftarrow \boldsymbol{x}_k + \alpha_k\boldsymbol{p}_k$
    $\gamma_{k+1} \leftarrow \boldsymbol{w}^T\cos(\boldsymbol{\Omega}\boldsymbol{x}_{k+1} + \boldsymbol{b})$
    while $\gamma_{k+1} - \gamma_k - \frac{1}{2}\alpha_k\boldsymbol{r}_k^T\boldsymbol{p}_k > \varepsilon$ **do**
      $\alpha_k \leftarrow \frac{1}{2}\alpha_k$
      $\boldsymbol{x}_{k+1} \leftarrow \boldsymbol{x}_k + \alpha_k\boldsymbol{p}_k$
      $\gamma_{k+1} \leftarrow \boldsymbol{w}^T\cos(\boldsymbol{\Omega}\boldsymbol{x}_{k+1} + \boldsymbol{b})$
    end while
    $\boldsymbol{r}_{k+1} \leftarrow -\boldsymbol{\Omega}\text{diag}(\boldsymbol{w})\sin(\boldsymbol{\Omega}^T\boldsymbol{x}_{k+1} + \boldsymbol{b})$
    $\tau_k \leftarrow (\boldsymbol{r}_{k+1}^T\boldsymbol{r}_{k+1})(\boldsymbol{r}_k^T\boldsymbol{r}_k)^{-1}$
    $\boldsymbol{p}_{k+1} \leftarrow -\boldsymbol{r}_{k+1} + \tau_k\boldsymbol{p}_k$
    $k \leftarrow k + 1$
  end while
  return $\boldsymbol{x}_k$
end procedure

Algorithm 10 Optimization using the BFGS method

procedure OPTIMBFGS($\boldsymbol{x}_0$, $\boldsymbol{w}$, $\boldsymbol{\Omega}$, $\boldsymbol{b}$, ε)
  $\boldsymbol{r}_0 \leftarrow -\boldsymbol{\Omega}\text{diag}(\boldsymbol{w})\sin(\boldsymbol{\Omega}^T\boldsymbol{x}_0 + \boldsymbol{b})$
  $\gamma_0 \leftarrow \boldsymbol{w}^T\cos(\boldsymbol{\Omega}\boldsymbol{x}_k + \boldsymbol{b})$
  $\boldsymbol{B}_0 \leftarrow \boldsymbol{I}_n$
  $k \leftarrow 0$
  while $||\boldsymbol{r}_k||_2\, n^{-\frac{1}{2}} > \varepsilon$ **do**
    $\alpha_k \leftarrow 1$
    $\boldsymbol{p}_k \leftarrow -\boldsymbol{B}_k\boldsymbol{r}_k$
    $\boldsymbol{x}_{k+1} \leftarrow \boldsymbol{x}_k + \alpha_k\boldsymbol{p}_k$
    $\gamma_{k+1} \leftarrow \boldsymbol{w}^T\cos(\boldsymbol{\Omega}\boldsymbol{x}_{k+1} + \boldsymbol{b})$
    while $\gamma_{k+1} - \gamma_k - \frac{1}{2}\alpha_k\boldsymbol{r}_k^T\boldsymbol{p}_k > \varepsilon$ **do**
      $\alpha_k \leftarrow \frac{1}{2}\alpha_k$
      $\boldsymbol{x}_{k+1} \leftarrow \boldsymbol{x}_k + \alpha_k\boldsymbol{p}_k$
      $\gamma_{k+1} \leftarrow \boldsymbol{w}^T\cos(\boldsymbol{\Omega}\boldsymbol{x}_{k+1} + \boldsymbol{b})$
    end while
    $\boldsymbol{r}_{k+1} \leftarrow -\boldsymbol{\Omega}\text{diag}(\boldsymbol{w})\sin(\boldsymbol{\Omega}^T\boldsymbol{x}_{k+1} + \boldsymbol{b})$
    $\boldsymbol{\theta}_k \leftarrow \boldsymbol{x}_{k+1} - \boldsymbol{x}_k$
    $\boldsymbol{\kappa}_k \leftarrow \boldsymbol{r}_{k+1} - \boldsymbol{r}_k$
    $\boldsymbol{B}_{k+1} \leftarrow (\boldsymbol{I}_n + (\boldsymbol{\kappa}_k\boldsymbol{\theta}_k^T)(\boldsymbol{\kappa}_k^T\boldsymbol{\theta}_k)^{-1})^T\boldsymbol{B}_k(\boldsymbol{I}_n + (\boldsymbol{\kappa}_k\boldsymbol{\theta}_k^T)(\boldsymbol{\kappa}_k^T\boldsymbol{\theta}_k)^{-1}) + (\boldsymbol{\theta}_k\boldsymbol{\theta}_k^T)(\boldsymbol{\kappa}_k^T\boldsymbol{\theta}_k)^{-1}$
    $k \leftarrow k + 1$
  end while
  return $\boldsymbol{x}_k$
end procedure

Characteristic operations	GD	NGD	NFR	BFGS
Initial values		3		4
Inner loop	5 - 17	7 - 14	6 - 19	6 - 21
Backtracking	10 - 14		10 - 14	11 - 15
Function evaluation	3, 9, 13		3, 9, 13	3, 10, 14
Gradient calculation	2, 15	2, 12	2, 15	2, 16
Step size selection	6, 11	4	7, 11	7, 12
Search direction	7		4, 17	8
Trajectory	8, 12	10, 11	8, 12	9, 13
Convergence test	5	7	6	6

Table 7-2: Characteristic operations for the optimization step. The numbers in this table refer to line numbers in Algorithms 7 to 10, which represent the GD, NGD, NFR and BFGS implementations respectively.

7-2-3 Sequential Implementation

Similar to the RFE fitting implementations, most of the discussed operations can be performed straightforwardly by one of the basic linear algebra subprograms (BLAS) in the CBLAS library. For instance the vector update at line 8 of Algorithm 7 is a so-called "axpy" operation and the search direction computation at line 8 of Algorithm 10 is a symmetric matrix-vector multiplication.

Not all operations can be performed that straightforwardly however. For instance the BFGS update at line 19 of Algorithm 10 can be performed using

$$\begin{aligned}
\boldsymbol{B}_{k+1} &= \left(\boldsymbol{I}_n - \frac{\boldsymbol{\kappa}_k\boldsymbol{\theta}_k^T}{\boldsymbol{\theta}_k^T\boldsymbol{\kappa}_k}\right)\boldsymbol{B}_k\left(\boldsymbol{I}_n - \frac{\boldsymbol{\theta}_k\boldsymbol{\kappa}_k^T}{\boldsymbol{\theta}_k^T\boldsymbol{\kappa}_k}\right) + \frac{\boldsymbol{\kappa}_k\boldsymbol{\kappa}_k^T}{\boldsymbol{\theta}_k^T\boldsymbol{\kappa}_k} \\
&= \boldsymbol{B}_k - \boldsymbol{B}_k\frac{\boldsymbol{\theta}_k\boldsymbol{\kappa}_k^T}{\boldsymbol{\theta}_k^T\boldsymbol{\kappa}_k} - \frac{\boldsymbol{\kappa}_k\boldsymbol{\theta}_k^T}{\boldsymbol{\theta}_k^T\boldsymbol{\kappa}_k}\boldsymbol{B}_k + \frac{\boldsymbol{\kappa}_k\boldsymbol{\theta}_k^T}{\boldsymbol{\theta}_k^T\boldsymbol{\kappa}_k}\boldsymbol{B}_k\frac{\boldsymbol{\theta}_k\boldsymbol{\kappa}_k^T}{\boldsymbol{\theta}_k^T\boldsymbol{\kappa}_k} + \frac{\boldsymbol{\kappa}_k\boldsymbol{\kappa}_k^T}{\boldsymbol{\theta}_k^T\boldsymbol{\kappa}_k} \\
&= \boldsymbol{B}_k - \frac{\boldsymbol{c}_k\boldsymbol{\kappa}_k^T}{\boldsymbol{\theta}_k^T\boldsymbol{\kappa}_k} - \frac{\boldsymbol{\kappa}_k\boldsymbol{c}_k^T}{\boldsymbol{\theta}_k^T\boldsymbol{\kappa}_k} + \frac{\boldsymbol{\kappa}_k\boldsymbol{\theta}_k^T\boldsymbol{c}_k\boldsymbol{\kappa}_k^T}{\boldsymbol{\theta}_k^T\boldsymbol{\kappa}_k\boldsymbol{\theta}_k^T\boldsymbol{\kappa}_k} + \frac{\boldsymbol{\kappa}_k\boldsymbol{\kappa}_k^T}{\boldsymbol{\theta}_k^T\boldsymbol{\kappa}_k} \\
&= \boldsymbol{B}_k - \frac{1}{\boldsymbol{\theta}_k^T\boldsymbol{\kappa}_k}\boldsymbol{c}_k\boldsymbol{\kappa}_k^T - \frac{1}{\boldsymbol{\theta}_k^T\boldsymbol{\kappa}_k}\boldsymbol{\kappa}_k\boldsymbol{c}_k^T + \frac{\boldsymbol{\theta}_k^T\boldsymbol{c}_k}{\boldsymbol{\theta}_k^T\boldsymbol{\kappa}_k\boldsymbol{\theta}_k^T\boldsymbol{\kappa}_k}\boldsymbol{\kappa}_k\boldsymbol{\kappa}_k^T + \frac{1}{\boldsymbol{\theta}_k^T\boldsymbol{\kappa}_k}\boldsymbol{\kappa}_k\boldsymbol{\kappa}_k^T \qquad (7\text{-}9) \\
&= \boldsymbol{B}_k - \gamma\boldsymbol{c}_k\boldsymbol{\kappa}_k^T - \gamma\boldsymbol{\kappa}_k\boldsymbol{c}_k^T + \gamma\tau\boldsymbol{\kappa}_k\boldsymbol{\kappa}_k^T \\
&= \boldsymbol{B}_k + \gamma\tau\left(-\frac{1}{\tau}\boldsymbol{c}_k\boldsymbol{\kappa}_k^T - \frac{1}{\tau}\boldsymbol{\kappa}_k\boldsymbol{c}_k^T + \boldsymbol{\kappa}_k\boldsymbol{\kappa}_k^T\right) \\
&= \boldsymbol{B}_k + \gamma\tau\left(\boldsymbol{\kappa}_k - \frac{1}{\tau}\boldsymbol{c}_k\right)\left(\boldsymbol{\kappa}_k - \frac{1}{\tau}\boldsymbol{c}_k\right)^T - \frac{\gamma}{\tau}\boldsymbol{c}_k\boldsymbol{c}_k^T,
\end{aligned}$$

where the substitutions are denoted by

$$\boldsymbol{c}_k = \boldsymbol{B}_k \boldsymbol{\theta}_k, \quad \gamma = \frac{1}{\boldsymbol{\theta}_k^T \boldsymbol{\kappa}_k}, \quad \tau = \frac{\boldsymbol{\theta_k}^T \boldsymbol{c}_k}{\boldsymbol{\theta}_k^T \boldsymbol{\kappa}_k} + 1.$$

This way the BFGS update (6-24) is reduced to a combination of a symmetric matrix vector-multiplication, an "axpy" operation, two symmetric rank 1 matrix updates, two dot products and a few scalar operations. Another example is the vector update at line 17 of Algorithm 9. This can only be performed using

$$\boldsymbol{p}_{k+1} = \tau_k(-\tau_k^{-1}\boldsymbol{r}_{k+1} + \boldsymbol{p}_k), \tag{7-10}$$

which is a combination of an "axpy" operation and a scalar-vector multiplication. In all optimization implementations there are more of these situations, however these examples are sufficient to describe how to deal with these situations.

Besides the operations that require a combination of BLAS and scalar operations, there are also some operations that need to be programmed manually. In all optimization implementations, there are two situations like this.

The first situation is the gradient calculation at lines 2 and 15 of Algorithm 7 and also present in all other optimization implementations. This operation contains a sine computation and a diag() operator, which are not part of the CBLAS library. Hence the gradient calculation needs to be performed using

$$\boldsymbol{\delta} \leftarrow \boldsymbol{\Omega}^T \boldsymbol{x}_k + \boldsymbol{b}, \tag{7-11}$$

$$\delta_j \leftarrow w_j \sin(\delta_j) \quad \forall \quad j \in \{1, \ldots, m\}, \tag{7-12}$$

$$\boldsymbol{r}_k \leftarrow -\boldsymbol{\Omega}\boldsymbol{\delta}, \tag{7-13}$$

where the first part (7-11) can be performed using a combination of BLAS, the second part (7-12) has to be performed using a for-loop and the third part (7-13) can be performed using a straightforward matrix-vector multiplication. Here $\boldsymbol{\delta}$ is a support vector.

The other situation in which manual programming is required, is the function evaluation at lines 8, 9 and 13 of Algorithm 7 and also present in all other optimization implementations except for the NGD implementation. This operation contains a cosine computation, which is not part of the CBLAS library. Hence the function evaluation needs to be performed using

$$\boldsymbol{\delta} \leftarrow \boldsymbol{\Omega}^T \boldsymbol{x}_k + \boldsymbol{b}, \tag{7-14}$$

$$\delta_j \leftarrow \cos(\delta_j) \quad \forall \quad j \in \{1, \ldots, m\}, \tag{7-15}$$

$$\gamma_k \leftarrow \boldsymbol{w}^T \boldsymbol{\delta}, \tag{7-16}$$

where the first part (7-14) can be performed using a combination of BLAS, the second part (7-15) has to be performed using a for-loop and the third part (7-16) can be performed using a dot product. Here $\boldsymbol{\delta}$ is a support vector.

7-2-4 Parallel Implementation

For every command in the CBLAS library, there is a parallel equivalent in the CUBLAS library [26]. Therefore it is rather straightforward to turn the sequential implementations into parallel implementations. However the gradient computation and function evaluation need to be dealt with in a different fashion for parallel computing. The first part (7-11) and third part (7-13) of the gradient computation can be performed using CUBLAS straightforwardly, however for the second part (7-12) of the gradient computation the for-loop needs to be replaced by a kernel, performing the computation for each vector element at the same time. The same holds for the function evaluation (7-14, 7-15, 7-16).

Another important aspect of parallel computing is the data communication. As discussed in Section 4-2 parallel computing is the most effective when there is coarse-grained parallelism. Therefore all parallel optimization implementations first initialize $\boldsymbol{\Omega}$, $\boldsymbol{b}$, $\boldsymbol{w}$ and $\boldsymbol{x}_0$, then copy all data from CPU to GPU, then perform all computations and afterwards only copy $\boldsymbol{x}_k$ back, since that is the only interesting piece of information. No communication occurs in between of the computations. Note that copying scalar values is of no significance here, so this is not taken into account.

7-3 The DONE Algorithm

The DONE implementation is a combination of one of the RFE fitting implementations, one of the optimization implementations and an implementation of the initialization and exploration steps. Chapter 9 will reveal that the RLS implementation and the BFGS implementation perform the best, therefore these methods are used in the final DONE implementation.

7-3-1 Characteristic Operations

Most of the characteristic operations have already been discussed in this chapter. The characteristic operations for the DONE implementation that have not been discussed yet are listed below.

Drawing random numbers

There are several situations in which random numbers are drawn. In the initialization step the entries of $\boldsymbol{\Omega}$ and $\boldsymbol{b}$ are drawn from a normal and a uniform distribution respectively and in the exploration steps the entries of $\boldsymbol{\zeta}_i$ and $\boldsymbol{\xi}_i$ are drawn from a normal distribution.

Applying bounds

There are also several situations in which the bounds are applied in order to ensure that the solution stays within the domain $\mathcal{X}$. These situations are the exploration steps and during the optimization step. In this last case the optimization trajectory may cross the bounds, which should be prevented. The bounds can be applied using the min() and max() operators.

Stopping conditions

A characteristic of the DONE algorithm is that it has a fixed run time per cycle. This is ensured by running a fixed amount of iterations in the optimization step. So whether or not the optimum is reached, after this fixed amount of iterations the optimization step is terminated. Also the amount of backtracking iterations should be limited, to make sure that the backtracking procedure will terminate even if due to a numerical error the Wolfe conditions are never satisfied.

7-3-2 Mathematical Implementation

Algorithm 11 represents the final DONE implementation. Table 7-3 lists the characteristic operations and at which lines they are incorporated.

Characteristic operations	**Algorithm 11**	**Appendix B**
Initial values	2, 3	187 - 210
Outer loop	9 - 42	212 - 408
Taking measurements	10	215 - 217, 410 - 412
Domain conversion	11	54 - 69, 226 - 228
Update relations	13 - 15	230 - 247
Initial values	20	249 - 270
Inner loop	21 - 37	294 - 392
Backtracking	26 - 30	315 - 334
Function evaluation	19, 25, 29	71 - 91, 289, 290, 327, 328
Gradient calculation	18, 32	93 - 110, 281, 282, 349, 350
Step size selection	22, 27	300, 301, 319, 320
Search direction	23	297, 298
Trajectory	24, 28, 31	305 - 308, 322 - 325, 344 - 347
Convergence test	21	284 - 287, 294, 295, 352 - 355
Drawing random numbers	4, 5, 16, 38	181 - 185, 273, 274, 398, 399
Applying bounds	17, 31, 40	277, 278, 346, 347, 402, 403
Stopping conditions	30, 37	316, 317, 330, 331, 388, 389

Table 7-3: Characteristic operations for the DONE algorithm. The first block contains the characteristic operations for RFE fitting, the second block contains the characteristic operations for optimization and the third block contains the characteristic operations that are specific for the DONE algorithm. The numbers in this table refer to line numbers in Algorithm 11 and to line numbers in Appendix B. Characteristic operations that are not used are not listed here.

Note that in Algorithm 11 the final exploration step (during the last cycle) is not performed, ensuring that the final value of $\boldsymbol{x}_i$ that is returned is the actual found optimum.

Algorithm 11 The DONE algorithm

procedure DONE(f, $\boldsymbol{x}_0$, $\boldsymbol{x}_{lb}$, $\boldsymbol{x}_{ub}$, λ, $\sigma_{\boldsymbol{\omega}}$, $\sigma_{\boldsymbol{\zeta}}$, $\sigma_{\boldsymbol{\xi}}$)
  $\boldsymbol{w}_0 \leftarrow \mathbf{0}_m$ ▷ INITIALIZATION (2 - 7)
  $\boldsymbol{P}_0 \leftarrow \lambda^{-1}\boldsymbol{I}_m$
  $\boldsymbol{\omega}_j \sim \mathcal{N}\left(\mathbf{0}_n, \sigma_{\boldsymbol{\omega}}^2 \boldsymbol{I}_n\right) \quad \forall \quad j \in \{1, \ldots, m\}$
  $b_j \sim \mathcal{U}\left(0, 2\pi\right) \quad \forall \quad j \in \{1, \ldots, m\}$
  $\boldsymbol{\Omega} \leftarrow [\boldsymbol{\omega}_1 \ \ \ldots \ \ \boldsymbol{\omega}_m]$
  $\boldsymbol{b} \leftarrow [b_1 \ \ \ldots \ \ b_m]^T$
  $k \leftarrow 0$
  for $i = 1 : N$ **do**
    $y_i \leftarrow f(\boldsymbol{x}_k)$ ▷ MEASUREMENT (10 - 11)
    $\boldsymbol{a}_i \leftarrow \cos(\boldsymbol{\Omega}^T \boldsymbol{x}_k + \boldsymbol{b})$
    $\gamma_i \leftarrow (1 + \boldsymbol{a}_i^T \boldsymbol{P}_{i-1} \boldsymbol{a}_i)^{-1}$ ▷ RFE FITTING (12 - 15)
    $\boldsymbol{g}_i \leftarrow \gamma_i \boldsymbol{P}_{i-1} \boldsymbol{a}_i$
    $\boldsymbol{w}_i \leftarrow \boldsymbol{w}_{i-1} + \boldsymbol{a}_i^T \boldsymbol{w}_{i-1} \boldsymbol{g}_i + y_i \boldsymbol{g}_i$
    $\boldsymbol{P}_i \leftarrow \boldsymbol{P}_{i-1} - \gamma_i^{-1} \boldsymbol{g}_i \boldsymbol{g}_i^T$
    $\boldsymbol{\zeta}_i \sim \mathcal{N}(\mathbf{0}_n, \sigma_{\boldsymbol{\zeta}}^2 \boldsymbol{I}_n)$ ▷ EXPLORATION (16 - 17)
    $\boldsymbol{x}_k \leftarrow \min(\max(\boldsymbol{x}_k + \boldsymbol{\zeta}_i, \boldsymbol{x}_{lb}), \boldsymbol{x}_{ub})$
    $\boldsymbol{r}_k \leftarrow -\boldsymbol{\Omega}\text{diag}(\boldsymbol{w}_i)\sin(\boldsymbol{\Omega}^T \boldsymbol{x}_k + \boldsymbol{b})$ ▷ OPTIMIZATION(18 - 37)
    $\gamma_k \leftarrow \boldsymbol{w}_i^T \cos(\boldsymbol{\Omega} \boldsymbol{x}_k + \boldsymbol{b})$
    $\boldsymbol{B}_k \leftarrow \boldsymbol{I}_n$
    while $||\boldsymbol{r}_k||_2 \, n^{-\frac{1}{2}} > \varepsilon$ **do**
      $\alpha_k \leftarrow 1$
      $\boldsymbol{p}_k \leftarrow -\boldsymbol{B}_k \boldsymbol{r}_k$
      $\boldsymbol{x}_{k+1} \leftarrow \boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k$
      $\gamma_{k+1} \leftarrow \boldsymbol{w}_i^T \cos(\boldsymbol{\Omega} \boldsymbol{x}_{k+1} + \boldsymbol{b})$
      while $\gamma_{k+1} - \gamma_k - \frac{1}{2}\alpha_k \boldsymbol{r}_k^T \boldsymbol{p}_k > \varepsilon$ **do**
        $\alpha_k \leftarrow \frac{1}{2}\alpha_k$
        $\boldsymbol{x}_{k+1} \leftarrow \boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k$
        $\gamma_{k+1} \leftarrow \boldsymbol{w}_i^T \cos(\boldsymbol{\Omega} \boldsymbol{x}_{k+1} + \boldsymbol{b})$
      end while
      $\boldsymbol{x}_{k+1} \leftarrow \min(\max(\boldsymbol{x}_{k+1}, \boldsymbol{x}_{lb}), \boldsymbol{x}_{ub})$
      $\boldsymbol{r}_{k+1} \leftarrow -\boldsymbol{\Omega}\text{diag}(\boldsymbol{w}_i)\sin(\boldsymbol{\Omega}^T \boldsymbol{x}_{k+1} + \boldsymbol{b})$
      $\boldsymbol{\theta}_k \leftarrow \boldsymbol{x}_{k+1} - \boldsymbol{x}_k$
      $\boldsymbol{\kappa}_k \leftarrow \boldsymbol{r}_{k+1} - \boldsymbol{r}_k$
      $\boldsymbol{B}_{k+1} \leftarrow (\boldsymbol{I}_n + (\boldsymbol{\kappa}_k \boldsymbol{\theta}_k^T)(\boldsymbol{\kappa}_k^T \boldsymbol{\theta}_k)^{-1})^T \boldsymbol{B}_k (\boldsymbol{I}_n + (\boldsymbol{\kappa}_k \boldsymbol{\theta}_k^T)(\boldsymbol{\kappa}_k^T \boldsymbol{\theta}_k)^{-1}) + (\boldsymbol{\theta}_k \boldsymbol{\theta}_k^T)(\boldsymbol{\kappa}_k^T \boldsymbol{\theta}_k)^{-1}$
      $k \leftarrow k + 1$
    end while
    if $i < N$ **then**
      $\boldsymbol{\xi}_i \sim \mathcal{N}(\mathbf{0}_n, \sigma_{\boldsymbol{\xi}}^2 \boldsymbol{I}_n)$ ▷ EXPLORATION (39 - 40)
      $\boldsymbol{x}_k \leftarrow \min(\max(\boldsymbol{x}_k + \boldsymbol{\xi}_i, \boldsymbol{x}_{lb}), \boldsymbol{x}_{ub})$
    end if
  end for
  return $\boldsymbol{x}_k$
end procedure

7-3-3 Sequential Implementation

The sequential DONE implementation already exists and is tuned such that it has very good performance properties. The difference between the existing sequential DONE implementation and the implementation as shown in Algorithm 11 is that the existing implementation consists of the IQR method combined with the LBFGS method. Furthermore the existing implementation also makes use of the CBLAS library and a specific library for drawing random numbers. A final difference is that the sequential implementation works with double-precision floating-points whereas the parallel implementation works with single-precision floating-points.

7-3-4 Parallel Implementation

The parallel implementation of the DONE algorithm can be constructed by combining the parallel implementations of the RLS method and the BFGS method. Only the characteristic operations in the third block in Table 7-3 need to be implemented from scratch.

For drawing random numbers, there is a dedicated library called CURAND [28]. This library allows drawing large amounts of random numbers with the benefit of parallel computing. It should be kept in mind though that the normal probability distribution engine can only draw even amounts of random numbers [28]. For applying the bounds, for instance at line 31 of Algorithm 11, a kernel needs to be constructed to perform

$$x_j \leftarrow \min(\max(x_j, x_{lbj}), x_{ubj}) \quad \forall \quad j \in \{1, \ldots, n\} \tag{7-17}$$

for each vector element at the same time.

Another important aspect of parallel computing is the data communication. As discussed in Section 4-2 parallel computing is the most effective when there is coarse-grained parallelism. Therefore the communication is kept at a minimum. The parallel DONE implementation initializes $\boldsymbol{\Omega}$, $\boldsymbol{b}$, $\boldsymbol{w}_0$, and $\boldsymbol{P}_0$ on the GPU, so the only communication between CPU and GPU is copying $\boldsymbol{x}_i$ back and forth in order to take measurements. Note that copying scalar values is of no significance here so this is not taken into account.

Chapter 8

Experiments

In order to determine which of the methods for the RFE fitting step and for the optimization step perform best, several experiments are to be performed. This chapter describes these experiments and also which criteria can be used to conclude something about the performance of the selected methods. The experiments that are to be performed on the DONE algorithm itself are discussed as well.

8-1 Dimensions

There are three dimensions of significance for the DONE algorithm. The first is the input dimension n of the objective function, the second is the amount of Fourier features m and the third is the amount of measurements N. Since N only determines how many cycles of the algorithm are performed, no parallel benefit is to be obtained by increasing N. By increasing n and m however, the suitability for parallelism can be improved. Since generally $N > n$ and $m > n$ holds, the dimension which will reach the largest values is m, which is hence chosen to be the independent parameter. In order to obtain results over a wide range of dimensions, the values for m are chosen from

$$2^j \quad \forall \quad j \in \{1, \ldots, 15\}$$

and in order for the experiments to be comparable, a direct relation must exist between m and n, which is chosen to be

$$n = \text{ceil}\left(\frac{m}{44}\right), \tag{8-1}$$

where the ceil() operator rounds the result up to the nearest integer. This appears to be a rather randomly chosen relation, however it ensures that $N > n$ for a fixed value of $N = 1000$ and all chosen values for m. Table 8-1 lists all combinations of dimensions that are used in the experiments.

#	1	2	3	4	5	6	7	8
n	1	1	1	1	1	2	3	6
m	2	4	8	16	32	64	128	256
N	1000	1000	1000	1000	1000	1000	1000	1000

#	9	10	11	12	13	14	15
n	12	24	47	94	187	373	745
m	512	1024	2048	4096	8192	16384	32768
N	1000	1000	1000	1000	1000	1000	1000

Table 8-1: Dimensions for experiments

8-2 Criteria

The experiment results are to be judged based on several criteria. The most important criterion is of course the run time. The ultimate goal is that the fastest parallel implementation is faster than the fastest sequential implementation. However also the accuracy of the results is an important criterion. A very fast implementation is of no use if it does not yield the correct results. Furthermore some of the implementations require a certain amount of iterations and a certain amount of backtracking iterations. These are also pieces of information that are of importance. The important criteria for the experiments are:

- Obtained accuracy
- Run time
- Amount of (backtracking) iterations

The expectation for these criteria is that the accuracy and the amount of (backtracking) iterations of the parallel implementations are similar to those of their sequential counterparts and that the run time of the parallel and sequential implementations has a profile similar to that in Figure 4-2. An important remark is that though the DONE algorithm is specifically designed for noisy, expensive objective functions without derivatives, all functions that are used for the experiments are smooth, low-cost and derivatives are available. However for comparing the implementations this does not matter, as long as the derivatives are not used of course. For the final experiment the sequential and parallel DONE implementations are to be tested on a practical application which is noisy, expensive and derivative-free.

8-3 Experiments

For the RFE fitting step, the optimization step and the full DONE algorithm, several experiments can be designed that test the acquired implementations for all dimensions listed in Table 8-1 in order to say something about the criteria listed above.

8-3-1 RFE Fitting

In order to test the implementations for the RFE fitting step, a surrogate function has to be constructed for a certain known function. This function has to be easily scalable, whilst maintaining its characteristics. The inverted Gaussian function $G : \mathbb{R}^n \mapsto \mathbb{R}$, defined by

$$G(\boldsymbol{x}) = -e^{-\frac{1}{n}\boldsymbol{x}^T\boldsymbol{x}}, \tag{8-2}$$

is very suitable for this case. By dividing the exponent by n the function maintains its characteristics within the domain $\mathcal{X}$, even for very large values of n.

Similar to the DONE algorithm itself, the entries of $\boldsymbol{\Omega}$ and $\boldsymbol{b}$ are drawn from a normal and a uniform probability distribution respectively. However the measurement points $\boldsymbol{x}_i$ are also drawn from a uniform probability distribution in this case, in such a way that they are all located within $\mathcal{X}$.

The obtained accuracy can be measured by computing the root-mean-square (RMS) value of the output error, which is denoted by

$$\epsilon = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(G(\boldsymbol{x}_i) - \hat{f}(\boldsymbol{x}_i))^2}, \tag{8-3}$$

where $\hat{f} : \mathbb{R}^n \mapsto \mathbb{R}$ is the constructed surrogate function and $\{\boldsymbol{x}_i \in \mathcal{X}\}_{i=1\ldots N}$ can either be the set of measurement locations used to construct the surrogate function, which results in the so-called training error, or a new set of measurement locations, which results in the so-called testing error.

The run time can be measured in milliseconds using the command `clock()`. Also the amount of iterations and the amount of backtracking iterations can simply be measured.

8-3-2 Optimization

In order to test the implementations for the optimization step, a pre-constructed RFE has to be optimized. By drawing the entries of $\boldsymbol{\Omega}$ and $\boldsymbol{w}$ from a normal probability distribution and the entries of $\boldsymbol{b}$ from a uniform probability distribution, the RFE denoted by

$$\hat{f}(\boldsymbol{x}) = \boldsymbol{w}^T \cos(\boldsymbol{\Omega}^T\boldsymbol{x} + \boldsymbol{b}) \tag{8-4}$$

can be constructed, which has many local optima at completely random locations. Since the actual locations of these optima are unknown, it it not possible to measure the exact obtained accuracy. However, since all experiments start from $\boldsymbol{x}_0 = \boldsymbol{0}_n$, by computing the RMS value of the argument of the found optimum, some information about the found optimum can be acquired.

The run time can be measured in milliseconds using the command `clock()`. Also the amount of iterations and the amount of backtracking iterations can simply be measured.

8-3-3 The DONE Algorithm

The best performing methods for the RFE fitting step and the optimization step are to be combined into an implementation that follows the procedure as described in Chapter 3. The resulting implementation can be tested using the inverted Gaussian function (8-2) as well. However, it may be useful to test it on a practical application as well, hence it is also to be tested using one of the applications the DONE algorithm was originally developed for, OBFN tuning [3].

Inverted Gaussian

When testing the parallel DONE implementation on an inverted Gaussian function, the obtained accuracy can be measured by computing the RMS value of the argument of the found optimum. Since the actual optimum is located at zero, this RMS value is a proper measure of how accurate the found optimum is. The run time can be measured in milliseconds using the command `clock()`. Since the DONE algorithm uses a fixed amount of iterations in the optimization step, the amount of (backtracking) iterations is not of interest in this case. The parameters that are used are listed in Table 8-2.

Besides global information about the run time it may also be interesting to find out which parts of the parallel implementation take the most time. The implementation can be divided into the following parts:

- Allocating and freeing memory
- Creating and destroying handles
- Memory transfer from host to device (H2D)
- Memory transfer from device to host (D2H)
- Initialization
- Taking measurements
- RFE fitting
- Optimization
- Exploration

Parameter	Gaussian experiments	OBFN experiment
Starting point	$\boldsymbol{x}_0 = 0.5 \cdot \mathbf{1}_n$	$\boldsymbol{x}_0 = 0.5 \cdot \mathbf{1}_n$
Lower bound	$\boldsymbol{x}_{lb} = -\mathbf{1}_n$	$\boldsymbol{x}_{lb} = \mathbf{0}_n$
Upper bound	$\boldsymbol{x}_{ub} = \mathbf{1}_n$	$\boldsymbol{x}_{ub} = \mathbf{1}_n$
Regularization	$\lambda = 0.001$	$\lambda = 1$
RFE frequency	$\sigma_{\boldsymbol{\omega}} = 1$	$\sigma_{\boldsymbol{\omega}} = 3$
Pre-exploration	$\sigma_{\boldsymbol{\zeta}} = 0.001$	$\sigma_{\boldsymbol{\zeta}} = 0.002$
Post-exploration	$\sigma_{\boldsymbol{\xi}} = 0.001$	$\sigma_{\boldsymbol{\xi}} = 0.002$

Table 8-2: Parameters for experiments on the DONE implementation

OBFN tuning

An optical beam forming network (OBFN) is used to process signals from different antenna elements in such a way that their phases are aligned. Actuators on the OBFN can be used to control the signal delays. If the desired delay is known, the problem of tuning the OBFN can be considered to be an optimization problem [3]. The objective to be minimized is the difference between the delay provided by the OBFN and the desired delay. This is useful for applications such as aircraft-satellite communication. The goal is to improve the signal-to-noise ratio of the incoming signal, which is converted from the electrical domain into the optical domain and processed using optical ring resonators (ORRs). ORRs can provide a tunable time delay to signals, but only over a small frequency band. Cascades of multiple ORRs can provide a constant delay over larger bandwidths [3], and this is where the large-scale aspect of OBFN tuning comes into play.

Contrary to the previously discussed experiments, the OBFN simulation is noisy, expensive and derivative-free, hence this is a good benchmark for comparing the parallel DONE implementation to the sequential DONE implementation on a real-life practical application.

When testing the DONE implementations on the OBFN tuning application, the obtained accuracy can be measured by running a computer simulation of OBFN tuning and taking the output value, which is the mean-squared value of the difference between the delay provided by the OBFN and the desired delay [3]. The run time can be measured in milliseconds using the command `clock()`. Since the DONE algorithm uses a fixed amount of iterations in the optimization step, the amount of (backtracking) iterations is not of interest in this case. What may be interesting though, is visualising how the objective value diminishes while the amount of measurements increases. The parameters that are used are listed in Table 8-2. For the OBFN experiment a specific set of dimensions is used, dimensions which are representative for a large-scale case of OBFN tuning, these dimensions are

$$n = 160, \quad m = 12000, \quad N = 50000,$$

which are larger than the dimensions used in [3].

Properties	**Set-up 1**	**Set-up 2**
CUDA version	7.5	7.5
CUDA compiler	NVCC, 64 bit	NVCC, 64 bit
C compiler	CL, 64 bit	GCC, 64 bit
Precision	Single	Single
Operating system	Microsoft® Windows 7, 64 bit	Microsoft® Windows 10, 64 bit
CPU type	Intel® Xeon® E5-1620 v3	Intel® Core™ i7-4510U
GPU type	NVIDIA® GeForce® GTX 760	NVIDIA® GeForce® GTX 850M
GPU cores	1152	640
GPU memory	2048 MB	2048 MB

Table 8-3: Hardware used for experiments

8-4 Software and Hardware

All experiments are to be performed ten times on set-up 1 in Table 8-3, after which for all results a mean and a standard deviation can be computed. Set-up 2 is only used for verification of the results in Appendix A. The reason for using set-up 1 is the fact that the GPU in this set-up has a higher amount of cores and should hence be able to solve larger problems.

Chapter 9

Results

This chapter discusses the results of the experiments that are described in Chapter 8. These results can be used to determine the best performing methods for the RFE fitting step and for the optimization step. Moreover, after these best performing methods are combined into a final parallel implementation of the DONE algorithm, the results show how well the parallel implementation performs with respect to the existing sequential implementation.

Representation of results

Before the results can be analysed, the experiment data need to be preprocessed in order to remove unreliable results. Appendix A-3 discusses the preprocessing steps that are applied. Since all experiments are performed ten times, the figures in this chapter show the mean and the standard deviation for the results. As long as the standard deviation does not drastically surpass the mean, the results can be considered to be reliable.

9-1 RFE Fitting

Figures 9-1 to 9-6 show the experiment results for the RFE fitting step. As discussed in Section 8-3-1 the goal of each experiment is to obtain a RFE that fits the inverted Gaussian function (8-2). The outcomes of each experiment are the training error and testing error, the run time and the amount of iterations. Since the RLS method and the IQR method do not require any iterations, Figures 9-5d and 9-6d do not show any results. Some figures show all data concentrated in a small area, which leads to large white spaces. The results are presented in this way intentionally. To allow for easy comparison between results of different implementations, the axes in all corresponding figures have the same scale. Standard deviations that are out of scope are not plotted at all. Not all experiments are performed for the full set of dimensions listed in Table 8-1. This is either due to the fact that for the largest dimensions the run time becomes too large or due to the fact that for the 15^{th} set of dimensions the GPU memory is exceeded.

Accuracy

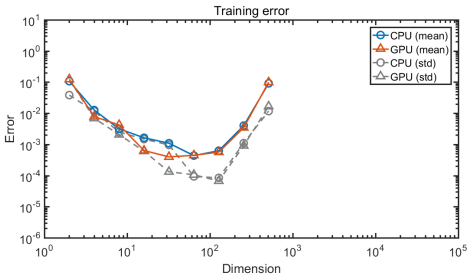
Figures 9-1a and 9-1b to 9-6a and 9-6b reveal that, as expected and discussed in Section 8-2, the training error and testing error results of the parallel implementations and those of the sequential implementations are very similar. Only for the IQR method this is not completely the case. Moreover, for all methods, the results have more or less the same profile. This is not a constant profile though, naturally the choice of the dimensions n, m and N has a large influence on how good the RFE fit turns out to be. Straightforwardly the used combinations of dimensions, as listed in Table 8-1, do not all yield the same accuracy. Nevertheless, the training error values are all below 1. It can be expected that the testing errors are larger than the training errors, however this only appears to occur for dimensions $m \geq 1000$. This means that for these dimensions the function fit becomes less reliable, so in order to acquire a better fit either N or m needs to be increased in these cases. However since all tested methods yield a similar training and testing error profile, the experiment results are suitable for further comparison.

Amount of iterations

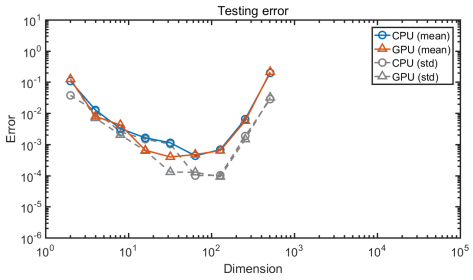
Figures 9-1d to 9-6d show that also for the amount of iterations, as expected and discussed in Section 8-2, the profiles of the parallel and sequential implementations are very similar. As can be expected, the GD method requires the largest amount of iterations, the NGD and NFR methods require more or less the same amount of iterations and the LFR method requires the smallest amount of iterations. Remarkable is that the amount of iterations does not increase drastically as the dimensions increase. One reason for this may be that the step size is chosen as $\alpha_i = \frac{1}{10} ||\mathbf{\Phi}_i||_F^{-1}$, which ensures that it is adapted after each measurement, thus keeping the amount of iterations at a minimum. Another reason may be that the parameter vector is not reset to $\boldsymbol{w}_k = \mathbf{0}_m$ each cycle, which allows the optimization to already start from a close-to-optimal point. A final reason may be that for the smallest dimensions $m \ll N$ holds, which means that after m measurements the least-squares problem becomes over-defined, thus yielding a very low iteration count for the remaining $N - m$ measurements. Only for results where $m \geq N$ this effect does no longer occur. Note that the shown amount of iterations is the total amount of iterations, not the amount of iterations per new measurement. Even though the amount of iterations does not increase as drastically as may be expected, the run time may still increase rapidly since the complexity of a single iteration increases linearly (for GD, NGD and NFR) or even quadratically (for LFR, RLS and IQR).

Run time

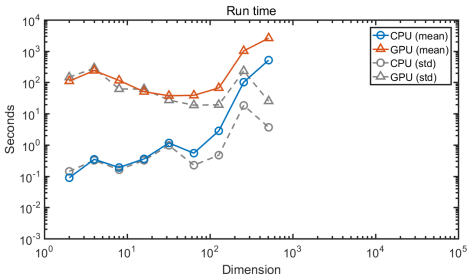
Figures 9-1c to 9-6c show the the run time results. For all methods, the results coincide with the behaviour that can be expected for any parallel implementation, as shown in Figure 4-2. For the GD, NGD and NFR methods, a crossover point seems to exist, however it is not yet reached within the dimensions used for the experiments. For the LFR and RLS methods, the crossover point is located around $m \approx 5000$. For the IQR method, the crossover point cannot be observed yet. The overall picture shows that the GD method is by far the slowest, followed by NGD and NFR, which perform more or less the same, followed by LFR and finally followed by RLS and IQR, where RLS appears to be performing best.



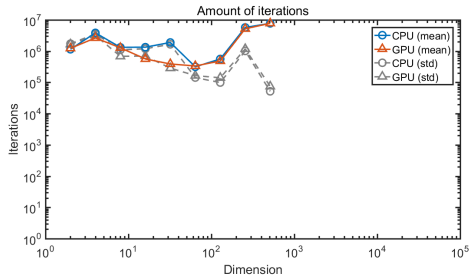
(a) Training error



(b) Testing error

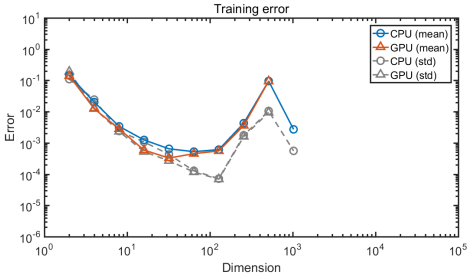


(c) Run time

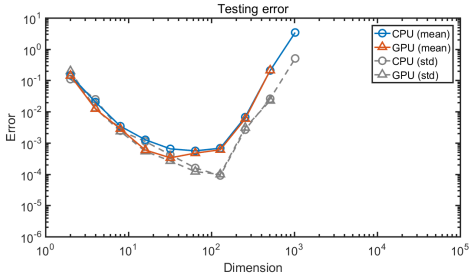


(d) Amount of iterations

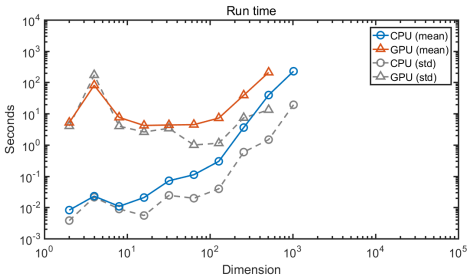
Figure 9-1: Experiment results for RFE fitting using the GD method. The solid lines represent the mean of ten performed experiments for each set of dimensions, the dashed lines represent the standard deviation. The dimension on the horizontal axis represents m, see Table 8-1 for its relation to n and N.



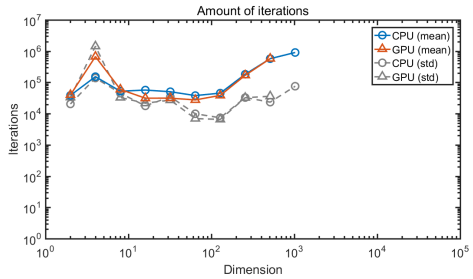
(a) Training error



(b) Testing error

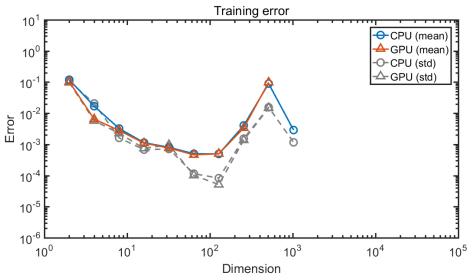


(c) Run time

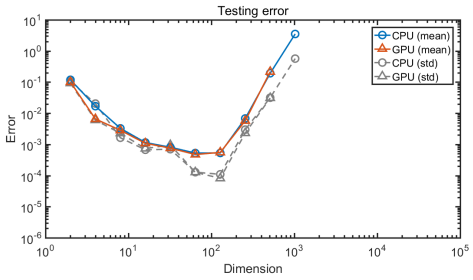


(d) Amount of iterations

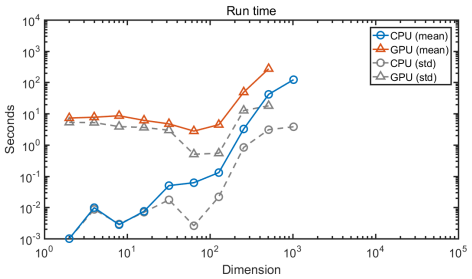
Figure 9-2: Experiment results for RFE fitting using the NGD method. The solid lines represent the mean of ten performed experiments for each set of dimensions, the dashed lines represent the standard deviation. The dimension on the horizontal axis represents m, see Table 8-1 for its relation to n and N.



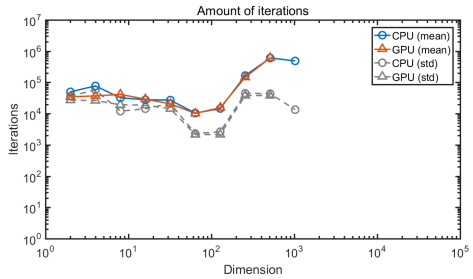
(a) Training error



(b) Testing error

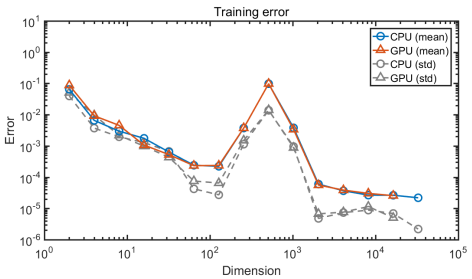


(c) Run time

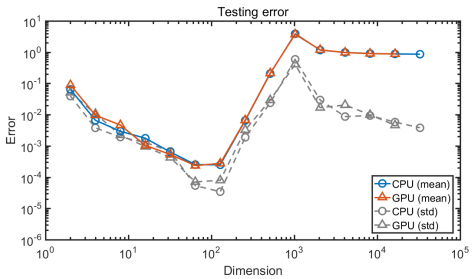


(d) Amount of iterations

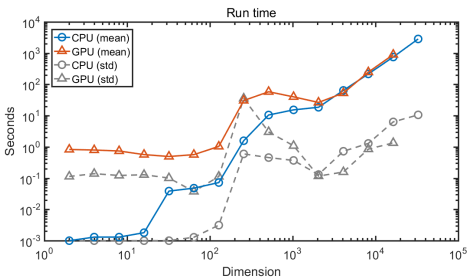
Figure 9-3: Experiment results for RFE fitting using the NFR method. The solid lines represent the mean of ten performed experiments for each set of dimensions, the dashed lines represent the standard deviation. The dimension on the horizontal axis represents m, see Table 8-1 for its relation to n and N.



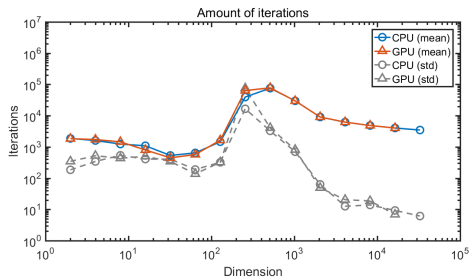
(a) Training error



(b) Testing error

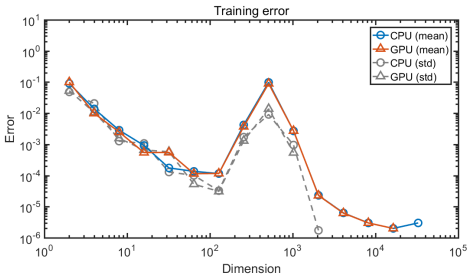


(c) Run time

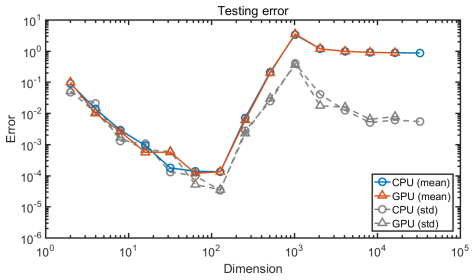


(d) Amount of iterations

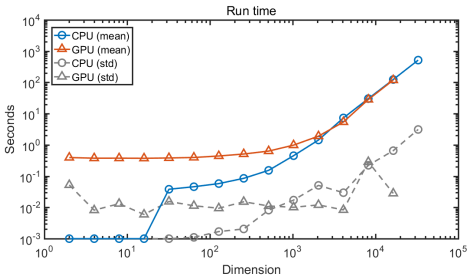
Figure 9-4: Experiment results for RFE fitting using the LFR method. The solid lines represent the mean of ten performed experiments for each set of dimensions, the dashed lines represent the standard deviation. The dimension on the horizontal axis represents m, see Table 8-1 for its relation to n and N.



(a) Training error



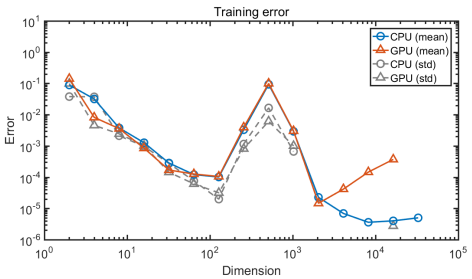
(b) Testing error



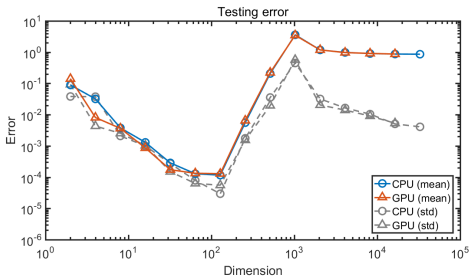
(c) Run time

(d) Amount of iterations

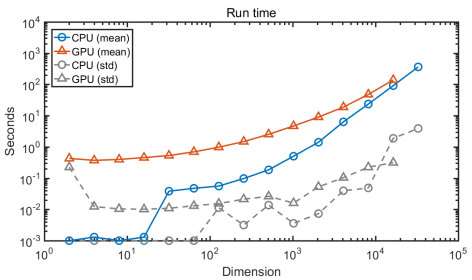
Figure 9-5: Experiment results for RFE fitting using the RLS method. The solid lines represent the mean of ten performed experiments for each set of dimensions, the dashed lines represent the standard deviation. The dimension on the horizontal axis represents m, see Table 8-1 for its relation to n and N.



(a) Training error



(b) Testing error

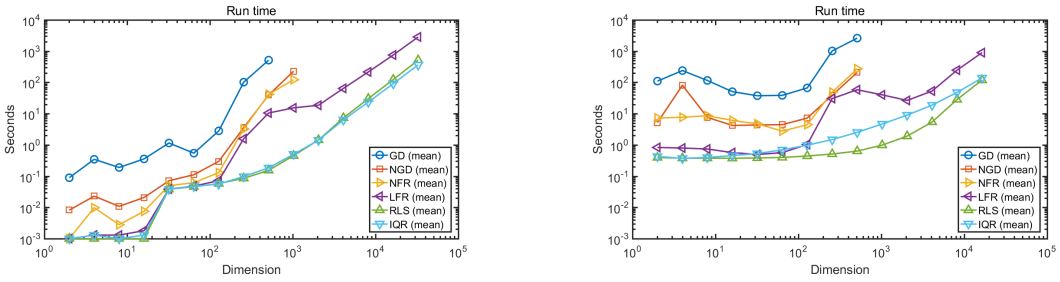


(c) Run time

(d) Amount of iterations

Figure 9-6: Experiment results for RFE fitting using the IQR method. The solid lines represent the mean of ten performed experiments for each set of dimensions, the dashed lines represent the standard deviation. The dimension on the horizontal axis represents m, see Table 8-1 for its relation to n and N.

Figure 9-7 shows the mean run times of all RFE fitting experiments. Figure 9-7a confirms the expected ranking for the sequential implementations, where GD is by far the slowest, followed by NGD and NFR, followed by LFR, and finally followed by RLS and IQR. The IQR method is, be it by a small difference, faster than the RLS method, confirming the choice of IQR in the existing DONE implementation. Figure 9-7b shows the same ranking for the parallel implementations, with the exception that the RLS method is faster than the IQR method in this case. Remarkable is that for both the sequential and the parallel case the LFR method can compete very well with the RLS and IQR methods for dimensions $m \leq 100$, only for higher dimensions it becomes significantly slower.



(a) Run time (CPU) **(b)** Run time (GPU)

Figure 9-7: Mean run times for all RFE fitting experiments. The dimension on the horizontal axis represents m, see Table 8-1 for its relation to n and N. No standard deviations are plotted in this figure.

Choice of RFE fitting method

Based on the fact that the parallel RLS method has the fastest run time for all tested dimensions, the RLS method is chosen as the RFE fitting that is to be used in the parallel DONE implementation. This does not mean though that it is not worth further investigating parallelization of the IQR method.

9-2 Optimization

Figures 9-8 to 9-11 show the experiment results for the optimization step. As discussed in Section 8-3-2 the goal of each experiment is to find a local optimum of a randomly established RFE. The outcomes of each experiment are the found optimum (RMS value of the argument), the run time, the amount of iterations and the amount of backtracking iterations. Since the NGD method does not require any backtracking iterations, Figure 9-9d does not show any results. Some figures show all data concentrated in a small area, which leads to large white spaces. The results are presented in this way intentionally. To allow for easy comparison between results of different implementations, the axes in all corresponding figures have the same scale. Standard deviations that are out of scope are not plotted at all. Not all experiments are performed for the full set of dimensions listed in Table 8-1. This is due to the fact that for the largest dimensions the run time becomes too large.

Accuracy

As discussed in Section 8-3-2 it is not possible to measure the actual accuracy in this experiment, since for the fully random RFE the locations of the optima are unknown. However it is known that all experiments start from $\boldsymbol{x}_0 = \mathbf{0}_n$, hence measuring the RMS value of the argument of the found optimum gives an indication of how far the optimization method has travelled. Hence, if all experiments yield a similar result, at least can be concluded that a fair comparison can be made.

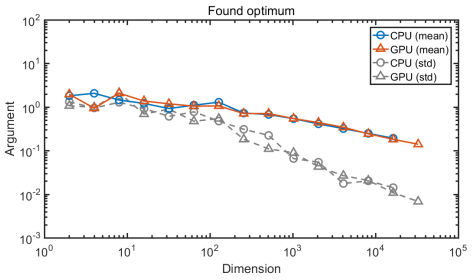
Figures 9-8a to 9-11a reveal a profile that is more or less the same for all methods except for the NFR method. Moreover, as expected and discussed in Section 8-2, the profile of the parallel implementations is very similar to that of the sequential implementations. Note that the results for the NFR method are very strange. For dimensions larger than $m \approx 100$, no results can be obtained any more. A possible explanation for this may be that the NFR framework may not operate in combination with the RFE framework very well. See Appendix A-1 for more information about this. For the GD, NGD and BFGS methods the found optimum increases slightly when the dimensions increase. Since the profile is very similar for all experiments except for the NFR experiments, at least the results for the GD, NGD and BFGS methods are suitable for further comparison.

Amount of iterations

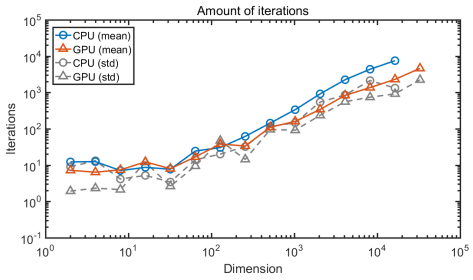
Figures 9-8b to 9-11b reveal that the amounts of iterations do not increase very drastically for increasing dimensions. This profile can, as expected and discussed in Section 8-2, be observed for the parallel implementations as well as for the sequential implementations. As can be expected, the GD method requires the most iterations and the BFGS method requires the least amount of iterations. The NGD method lies between these two methods and as discussed the NFR method does not really fit in this comparison due to its bad performance. It may be interesting to find out if the NGD method, which requires more iterations, may be able to compete in run time with the BFGS method, since the primary computational complexity of the NGD method increases linearly with increasing dimensions whereas the primary computational complexity of the BFGS method increases quadratically with increasing dimensions.

Amount of backtracking iterations

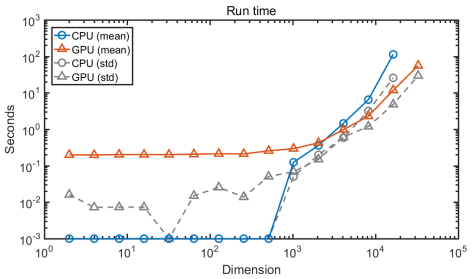
Figures 9-8d to 9-11d reveal a profile for the amounts of backtracking iterations which is very similar to that of the amounts of iterations. Also in this case this profile can, as expected and discussed in Section 8-2, be observed for the parallel implementations as well as for the sequential implementations. Note that the total amounts of backtracking iterations are shown, not the amount of backtracking iterations per iteration. Since the amounts of backtracking iterations are mostly approximately ten times higher than the amounts of iterations, this yields the conclusion that the amount of backtracking iterations remain below ten, which is an acceptable amount.



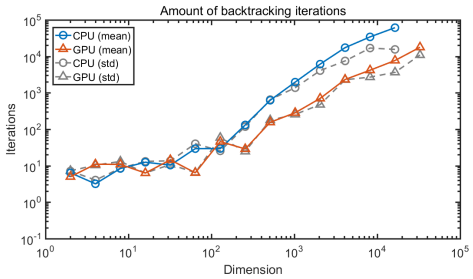
(a) Accuracy



(b) Amount of iterations

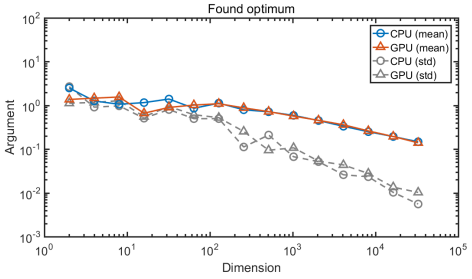


(c) Run time

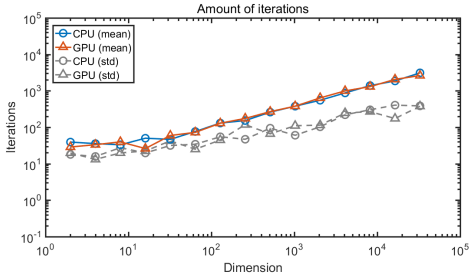


(d) Amount of backtracking iterations

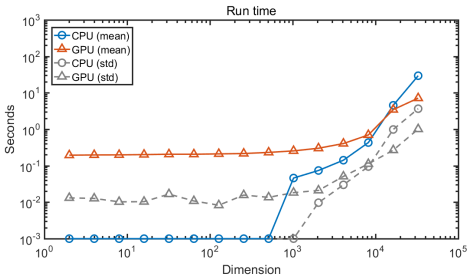
Figure 9-8: Experiment results for optimization using the GD method. The solid lines represent the mean of ten performed experiments for each set of dimensions, the dashed lines represent the standard deviation. The dimension on the horizontal axis represents m, see Table 8-1 for its relation to n and N.



(a) Accuracy



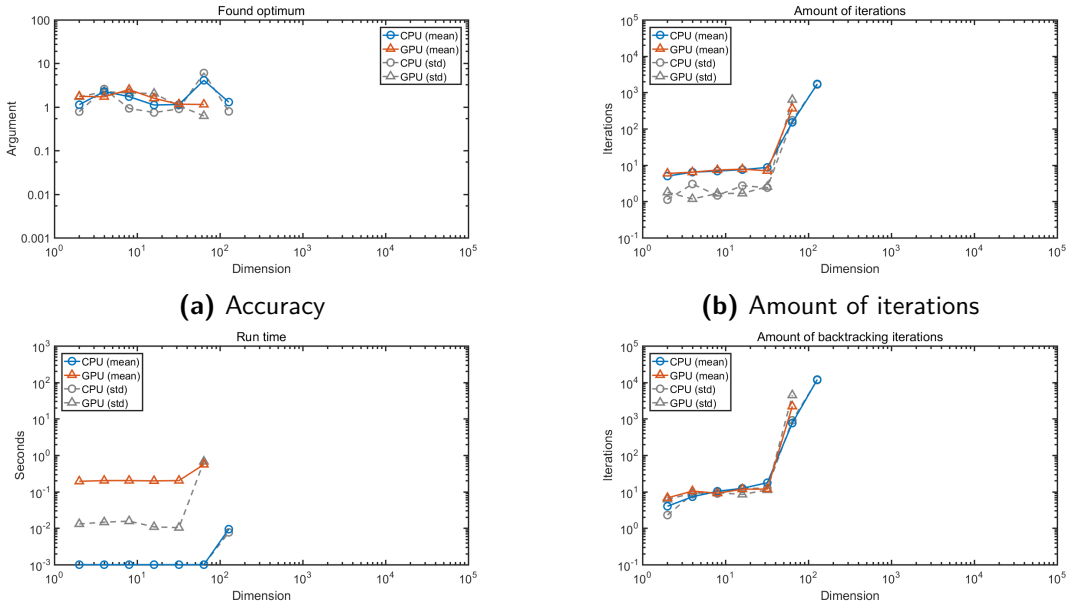
(b) Amount of iterations



(c) Run time

(d) Amount of backtracking iterations

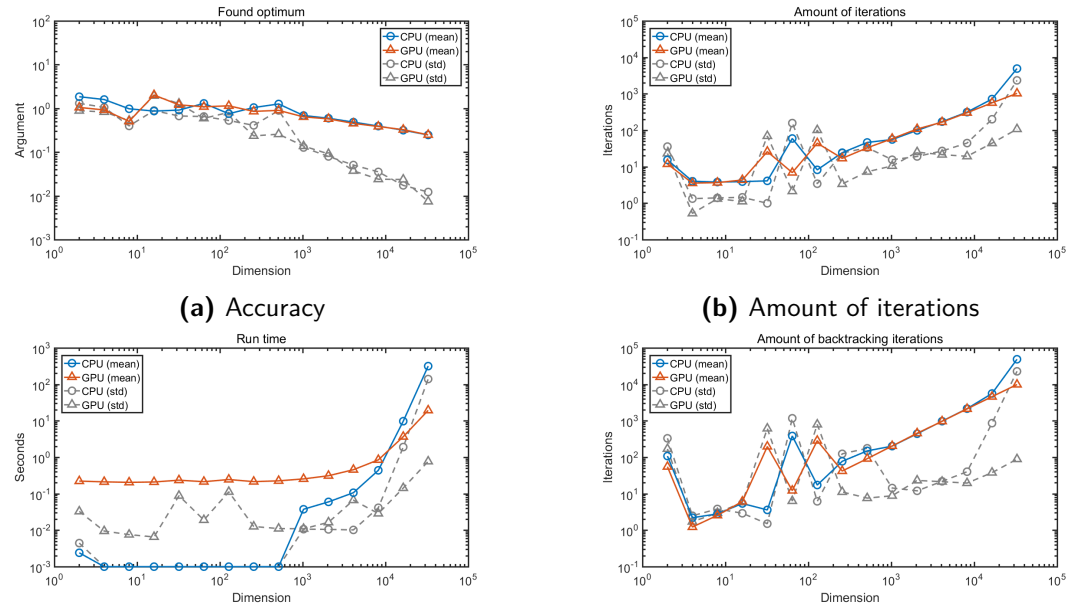
Figure 9-9: Experiment results for optimization using the NGD method. The solid lines represent the mean of ten performed experiments for each set of dimensions, the dashed lines represent the standard deviation. The dimension on the horizontal axis represents m, see Table 8-1 for its relation to n and N.



(a) Accuracy **(b)** Amount of iterations

(c) Run time **(d)** Amount of backtracking iterations

Figure 9-10: Experiment results for optimization using the NFR method. The solid lines represent the mean of ten performed experiments for each set of dimensions, the dashed lines represent the standard deviation. The dimension on the horizontal axis represents m, see Table 8-1 for its relation to n and N.



(a) Accuracy **(b)** Amount of iterations

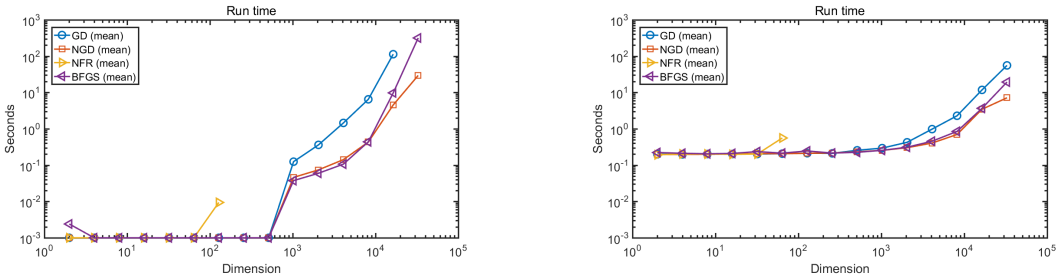
(c) Run time **(d)** Amount of backtracking iterations

Figure 9-11: Experiment results for optimization using the BFGS method. The solid lines represent the mean of ten performed experiments for each set of dimensions, the dashed lines represent the standard deviation. The dimension on the horizontal axis represents m, see Table 8-1 for its relation to n and N.

Run time

Figures 9-8c to 9-11c show the run time results. For all methods, except NFR, the results coincide with the behaviour that can be expected for any parallel implementation, as shown in Figure 4-2. The crossover points where the sequential implementations become slower than the parallel implementations are located at $m \approx 5000$ for the GD method and at $m \approx 10000$ for the NGD and BFGS methods. This indicates that the parallel GD implementation is most effective with respect to the degree of parallelism. However both the sequential and the parallel implementations of the GD method are slower than their respective NGD and BFGS counterparts. Another remarkable result is that the parallel NGD implementation and the parallel BFGS implementation perform more or less the same, so indeed the higher amount of iterations of the NGD method is compensated by its lower computational complexity. Even more remarkable is the fact that this is the case both for the sequential implementations and the parallel implementations. Hence the good performance of the NGD method is not a product of parallel programming but of very effective first-order optimization, which is of course the purpose of the NGD method [31, 33, 34].

Figure 9-12 shows the mean run times of all optimization experiments. Figure 9-12a confirms the discussed ranking for the sequential implementations, where GD is by far the slowest and NGD and BFGS perform more or less the same. The NFR method clearly cannot be taken into account here. Figure 9-12b shows the same ranking for the parallel implementations. For the highest dimensions NGD appears to be performing even better than BFGS, however since no results are available for higher dimensions, this cannot be confirmed.



(a) Run time (CPU) **(b)** Run time (GPU)

Figure 9-12: Mean run times for all optimization experiments. The dimension on the horizontal axis represents m, see Table 8-1 for its relation to n and N. No standard deviations are plotted in this figure.

Choice of optimization method

Since NGD and BFGS appear to have mostly the same performance with respect to the experiment criteria, the choice for one of them should be based on other criteria. In this case BFGS is chosen since this is a more established method for which more mathematical foundations exist. Moreover, when comparing rudimentary parallel DONE implementations using both the NGD and BFGS methods, the implementation containing the BFGS method appears to perform better, as can be seen in Appendix A-2.

9-3 The DONE Algorithm

As discussed in Section 8-3-3, the acquired parallel DONE implementation is to be tested on an inverted Gaussian function, using the dimensions in Table 8-1, and on a practical application called OBFN tuning, using dimensions that are suitable for this application.

9-3-1 Inverted Gaussian

Figure 9-13 shows the results of the first experiment involving the parallel DONE implementation. The goal of the experiment is to find the optimum of an inverted Gaussian function. The outcomes of the experiment are the found optimum (RMS value of the argument) and the run time.

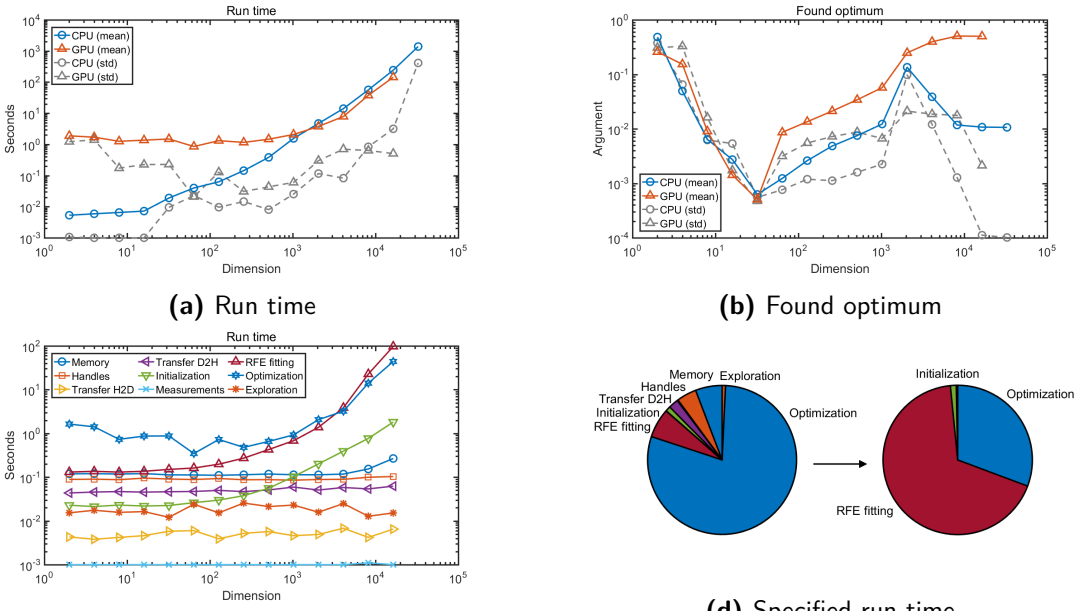
Accuracy

Figure 9-13b shows the obtained accuracy for the inverse Gaussian experiments. Since in this case the global optimum is known and located at $\boldsymbol{x}^* = \mathbf{0}_n$, the found optimum is a good measure for the accuracy. As can be seen for $m \leq 30$ the sequential implementation and the parallel implementation obtain more or less the same accuracy. Note that, as listed in Table 8-2, the starting point is $\boldsymbol{x}_0 = 0.5 \cdot \mathbf{1}_n$, so for the lowest dimensions the obtained accuracy is not so good. Of course this makes sense, since for these experiments a very small amount of Fourier features is used. For $m > 30$ the sequential implementation starts achieving better results than the parallel implementation. The parallel implementation even diverges at a certain point, this does not agree with the expectation that the obtained profiles should be similar, as discussed in Section 8-2. A possible explanation for this may be that the parameters are fixed throughout all experiments, and these parameters may not be optimal for all sets of dimensions used. However this is the case for both the parallel and the sequential implementation. Hence it appears that the parallel DONE implementation needs to be tuned separately from the sequential implementation.

Run time

Figure 9-13a shows the run time for the inverse Gaussian experiment. The profile is as can be expected similar to the profile in Figure 4-2. The parallel implementation is much slower for small dimensions, however for dimensions $m > 1000$ it starts outperforming the sequential implementation.

Figures 9-13c and 9-13d show the run time for the different parts of the parallel DONE implementation. The left pie chart in Figure 9-13d represents the first of the experiments whereas the right pie chart represents the last of the experiments. As can be seen the optimization step requires by far the largest amount of run time. The only parts for which the run time increases significantly when the dimensions increase are the initialization step, the optimization step and the RFE fitting step. As discussed in Section 4-2, for small dimensions the memory copies take up a significant part of the run time whereas for large dimensions they do not. Remarkable is that copying data from device to host requires much more run time than copying data from host to device.



(a) Run time

(b) Found optimum

(c) Specified run time

(d) Specified run time

Figure 9-13: Experiment results for Gaussian optimization using the DONE algorithm. The solid lines represent the mean of ten performed experiments for each set of dimensions, the dashed lines represent the standard deviation. The dimension on the horizontal axis represents m, see Table 8-1 for its relation to n and N.

9-3-2 OBFN Tuning

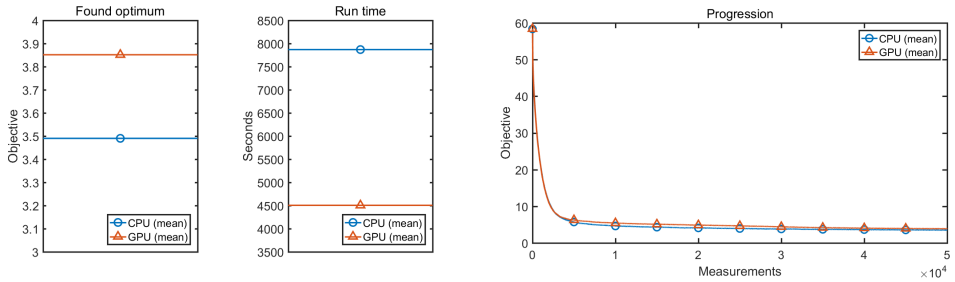
The most informative experiment is of course the OBFN experiment, since this yields how successful the parallel DONE implementation is on a practical application. Figure 9-14 shows the experiment results. As discussed in Section 8-3-3 the goal of the experiment is to optimize the outcome of a computer simulation of OBFN tuning. The outcomes of the experiment are the reached objective function value and the run time. The parameters listed in Table 8-2 are used.

Accuracy

Figure 9-14c shows the obtained accuracy for each cycle of the DONE algorithm. Figure 9-14a shows the final obtained accuracy for the OBFN experiment. As can be seen, for this specific experiment, the sequential DONE implementation reaches a higher accuracy than the parallel DONE implementation. This may be explained by the fact that the parallel DONE implementation works with single-precision floating-points whereas the sequential DONE implementation works with double-precision floating-points. Moreover there are some differences between the IQR and LBFGS methods used in the sequential implementation and the RLS and BFGS methods used in the parallel implementation, which may also cause a difference in accuracy. However, when looking at Figure 9-14c, which shows the reached objective values throughout running the DONE algorithm, the difference in accuracy is not that much, considering that both implementations start off with an objective value of around $f(\boldsymbol{x}_0) \approx 60$.

Run time

Figure 9-14b shows the run time for the OBFN experiment. As can be seen, for this specific experiment, the parallel DONE implementation is almost twice as fast as the sequential DONE implementation.



(a) Accuracy **(b)** Run time **(c)** Progression

Figure 9-14: Experiment results for OBFN tuning using the DONE algorithm. The solid lines represent the mean of ten performed experiments, the standard deviation is left out here because it is orders of magnitude smaller than the mean values. Note that there are more measurements than the amount of markers in Figure c.

Result	**Sequential**	**Parallel**
Objective value	3.50	3.85
Run time in seconds	7.87×10^3	4.50×10^3
Run time in hours	2:11	1:15

Table 9-1: Final experiment results

The numerical values of the final results shown in Figure 9-14 are listed in Table 9-1. From these values can be concluded that for this specific experiment the final achieved speed-up factor is

$$S = \frac{4.50 \times 10^3}{7.87 \times 10^3} = 1.75.$$

Chapter 10

Discussion

With all results available, the sub-questions that are discussed in Chapter 1 can be answered.

1. In which scenario is the DONE algorithm used and in which scenario can a parallel DONE implementation be useful?

Data-based online nonlinear extremum-seeker (DONE) is an algorithm that optimizes noisy and expensive objective functions which do not provide derivatives and which may be non-convex. It was developed for sensorless wavefront aberration correction in optical coherence tomography (OCT) [2] and tuning of an optical beam forming network (OBFN) [3]. OBFN tuning is a problem which may involve a large amount of parameters. In cases that the objective function has a large input dimension, a parallel implementation of the DONE algorithm may yield an improvement in run time.

2. What are the core elements of the DONE algorithm?

The core elements of the DONE algorithm are:

- RFE fitting, which comes down to solving a regularized linear least-squares problem in a recursive fashion [14].
- Optimization of a nonlinear surrogate function, where the surrogate function is established using RFE fitting [6].
- Exploration, which comes down to slightly perturbing the measurement locations in order to attempt global optimization.

Apart from these core elements, the initialization step of the DONE algorithm is also susceptible for parallel computing so, although it is not considered a core element, incorporating this into the parallel DONE implementation may help achieving a speed-up with respect to the sequential DONE implementation.

3. How can the DONE algorithm be classified and which other algorithms belong to the same classification?

The DONE algorithm can be classified as a model-based derivative-free optimization algorithm, despite the contrasting term "data-based" in its name. Examples of comparable methodology are NEWUOA [18] and Bayesian optimization [19]. NEWUOA and Bayesian optimization have some features which may be useful for the DONE algorithm as well, however these features are not specifically useful for parallel computing, so they are left out of the scope of this project.

4. What are the key concepts of parallel computing on a GPU using CUDA?

The power of a GPU lies in its large amount of cores, not in the individual capacity of the cores. Since a CPU has much less, however more powerful cores, parallel computing is only beneficial for large-scale computations. Also since communication between CPU and GPU takes a lot of time, only large-scale problems and coarse-grained problems may yield a speed-up since in these cases the amount of communication can be hidden behind the larger amount of computation. In order to make use of all capacities of parallel computing, the libraries CUBLAS and CURAND can be used to perform the core elements of the DONE algorithm.

5. What are the criteria that determine whether or not a numerical method is suitable for parallel computing?

The parallel computing criteria are:

- Computational complexity
- Degree of parallelism
- Required memory storage

All criteria are applied to the primary elements, which are inherent to the discussed methods, and the secondary elements, which are due to additional computations. The computational complexity and the required memory storage are presented as a function of the input dimension n and the amount of Fourier features m. The degree of parallelism is considered a qualitative criterion rather than a quantitative criterion.

6. What are the available numerical methods that can be used for these core elements and how do they compare with respect to the established criteria?

For the RFE fitting step, recursive methods and optimization-based methods can be used. The recursive least-squares (RLS), inverse QR (IQR), gradient descent (GD), Nesterov's accelerated gradient descent (NGD), nonlinear Fletcher-Reeves (NFR) and linear Fletcher-Reeves (LFR) methods are selected for parallel implementation based on the established criteria.

For the optimization step, first-order and second-order methods can be used. The gradient descent (GD), Nesterov's accelerated gradient descent (NGD), nonlinear Fletcher-Reeves (NFR) and Broyden-Fletcher-Goldfarb-Shanno (BFGS) methods are selected for parallel implementation based on the established criteria.

The initialization and exploration steps can be parallelized straightforwardly.

7. What are the criteria that are important for comparison between sequential and parallel implementations of the selected methods?

The experiment criteria are:

- Obtained accuracy
- Run time
- Amount of (backtracking) iterations

It is important to know what accuracy is obtained in order to be able to say something about the run time. It makes no sense to continue with an implementation that is very fast but obtains very inaccurate results. Not all implementations require iterations or backtracking iterations, however for those implementations that do it is interesting to measure the amount of (backtracking) iterations in order to verify whether they are fast or slow because they require complex computations or because they require many iterations.

8. Which of the selected methods perform best according to the established criteria?

The best performing parallel implementation for the RFE fitting step is the RLS method, closely followed by the IQR method. For the optimization step both the BFGS method and the NGD method appear to be the best performing parallel implementation.

RLS and IQR both obtain the same accuracy, but RLS has a faster run time. Both methods do not require any iterations. BFGS and NGD also obtain the same accuracy and although NGD requires more iterations, their run times are more or less the same. It is an interesting result that the first-order NGD method has the same run time as the second-order BFGS method and it may be useful to further investigate the possibilities of implementing the DONE algorithm using the NGD method.

9. How does the acquired parallel implementation of the DONE algorithm compare to the existing sequential implementation?

For the OBFN experiment, the parallel DONE implementation obtains an accuracy which is a minor bit worse than the accuracy obtained by the existing sequential implementation, however when considering the starting values for the objective value this difference in the final values is negligible. The parallel DONE implementation performs the OBFN experiment with a speed-up factor of $S = 1.75$ with respect to the sequential DONE implementation.

It should be kept in mind though, that the parallel implementation performs single-precision computations whereas the sequential implementation performs double-precision computations, which may also be the reason that the sequential implementation takes more run time. However, even if the sequential implementation would indeed be much faster if it also performed single-precision computations, as discussed in Section 4-6 there is much more promise for further improvement in GPU capacities than there is for further improvement in CPU capacities. Hence, even if a parallel implementation does not immediately yield a speed-up, this does not mean that parallelization should not be considered any further. So the obtained parallel DONE implementation may become even more useful in the future.

Chapter 11

Conclusion

11-1 Research Question

After answering the sub-questions, the main research question can be answered as well.

> What are the possibilities of parallelization of the DONE algorithm by means of implementing it on a GPU using CUDA, with the focus on achieving a maximum speed-up factor for large-scale problems, with respect to the existing sequential implementation, while maintaining correct results?

The answer in short is that by implementing the DONE algorithm on a GPU using the recursive least-squares (RLS) method to perform the RFE fitting step and the Broyden-Fletcher-Goldfarb-Shanno (BFGS) method to perform the optimization step, for optimization of a large-scale OBFN simulation, a speed-up factor of 1.75 can be obtained with respect to the sequential implementation, while remaining within reasonable bounds with respect to the correctness of the results. There is however no proof that this is the best possible speed-up factor. Several actions may be taken in order to possibly improve the results, as listed in the recommendations for future work.

11-2 Conclusions

Some more detailed conclusions can be drawn from the results:

- All parallel implementations behave as can be expected when it comes to run time. For small dimensions they are much slower than the sequential implementations, however as the dimensions increase the sequential implementations become much slower whereas the parallel implementations do not become much slower.

- Linear least-squares methods that are based on optimization cannot compete with dedicated linear least-squares methods with regard to run time, although the linear Fletcher-Reeves (LFR) method comes rather close.
- The first-order method Nesterov's accelerated gradient descent (NGD) does, for this application, a great job at competing with the second-order method Broyden-Fletcher-Goldfarb-Shanno (BFGS), not only in the parallel case but also in the sequential case.
- The Gaussian experiment results prove that the parallel DONE implementation can be faster than the sequential DONE implementation for large dimensions, however these results do not yet show maintained correctness of results.
- In the specific case of the OBFN experiment the parallel DONE implementation is definitely faster than the sequential implementation and in this case correctness of results is maintained.

11-3 Evaluation

The first question that comes to mind when evaluating this project, is whether or not the main research question has been answered. The main research question is too much of an open question to say something about this, since it cannot be proven that a maximum speed-up factor is achieved nor can it be proven that all possibilities have been explored. The most important part however, has been answered. One of the possibilities for obtaining a parallel implementation that is faster than the sequential implementation while maintaining correct results has been discovered.

Another question one may ask, is whether or not it makes a difference that the parallel DONE implementation works with single-precision variables whereas the sequential DONE implementation works with double-precision variables. If the sequential DONE implementation would be working with single-precision variables as well, would it be just as fast as the parallel DONE implementation? Of course this can only be determined by putting it to the test, however the fact that all sequential implementations of the separate RFE fitting and optimization methods work with single-precision as well and these sequential methods are all outperformed by their parallel counterparts eventually, suggests that the difference between single-precision and double-precision does not have that much influence on the run time.

A result that is very interesting is the specified run time in Figures 9-13c and 9-13d. It might have been beneficial to have these results for all performed experiments. Unfortunately this specified run time was only taken into account close to the end of the project.

An important thing for optimization methods is checking whether or not the found optimum is a minimum and not a maximum or a saddle point. This can be done by checking the positive-definiteness of the Hessian. This is not taken into account in all tested optimization implementations. The nature of the DONE algorithm is such that this is not very important, since every found point provides useful information, whether it is an actual optimum or not, however for verification of the final obtained optimum it may be beneficial to implement a Hessian check.

Another minor flaw in the development of the implementations is the step size selection for the optimization implementation of the NGD method. In the current implementation the step size is based on the maximum gradient value and not the maximum Hessian value, as it should be. Although the implementation works properly, it may be more efficient if the step size is selected based on the maximum Hessian value. Of course it would be a bit too much too actually perform a matrix-matrix multiplication in order to select a step size, but using the covariance information of the probability distribution of the frequency matrix may allow an improvement already.

Also the argumentation on which the choice of using the BFGS method instead of the NGD method is not very strong. It makes sense to use the BFGS method since it is a more established and theoretically faster method, however because the NGD method is so surprisingly fast and because more improvement may be possible, NGD should not be ignored.

During the literature survey for this project, a lot of parallel implementations of other algorithms were discovered. These implementations might have been a good example and also a good inspiration for how to parallelize the DONE algorithm. However as the project moved on, it turned more into an investigation into which methods best to use for a parallel DONE implementation instead of how to apply parallelization possibilities at a more detailed level. This is not necessarily a bad thing, the investigated matter in this master thesis is of much importance for obtaining a speed-up, however the more detailed parallelization possibilities should not be ignored and may be very useful for further parallelization.

A final improvement on the parallel DONE implementation, that is unfortunately not present in the implementation that is used for the experiments, is efficient surrogate function evaluation. In Algorithm 11 and in Appendix B this is implemented as efficiently as possible, however in the implementation used for the experiments the surrogate function is evaluated three times per iteration of the optimization step while it is only necessary to do this evaluation twice per iteration. Since the surrogate function evaluation cannot be considered a minor part of the code, this may mean that the final parallel DONE implementation is even faster than presented in the results.

11-4 Future Work

As discussed in Section 11-3 this master thesis focusses on which numerical methods to choose when trying to parallelize an algorithm. However there may be much more possibilities to speed-up the DONE algorithm on a more detailed level of parallelism. For instance the use of shared memory has not been explored yet, while using shared memory effectively may be able to speed-up the algorithm even more. As discussed in Section 11-3 there is a quite extensive amount of literature available describing how to implement certain numerical methods in parallel. These pieces of literature may describe more possibilities to obtain a speed-up.

Another possibility to obtain more speed-up is to use streams in order to add more concurrency. Although the steps of the DONE algorithm are mostly interdependent, some of them are not and thus they can be performed at the same time. This may yield some more speed-up. Also with regard to memory copies more concurrency can be added using streams. Some computations may already be started while data that is not yet needed is still being copied.

Moreover when problems become even more large-scale than the ones presented in this master thesis there may not be enough storage space on the GPU any more and concurrent memory copies become inevitable in order to maintain the gained speed-up.

As discussed in Appendix A-1 the reason that the NFR method does not yield proper results when optimizing the surrogate function (3-11) may be that it is not restarted at regular intervals. It may be interesting to find out if the NFR method with occasional restarts may perhaps behave just as well as the NGD method does.

Since the DONE algorithm performs a fixed amount of iterations per cycle, it is not really necessary to do the convergence check where the optimization is terminated once the gradient becomes sufficiently small. Omitting this part from the final parallel DONE implementation may yield a little bit more speed-up.

It is remarkable that the IQR method is such a superior method for sequential computing but that it performs worse when it comes to parallel computing. Maybe the way in which it is currently parallelized is not optimal. It may be useful to do further research into parallelization possibilities for the IQR method.

Since the choice for the BFGS method over the NGD method does not have very solid argumentation, it may be useful to do more research into if there could be a possibility for the NGD method to outperform the BFGS method. Especially since the DONE algorithm only performs a fixed amount of iterations, this may yield surprising results.

11-5 Final Note

As discussed more research can be performed to obtain the optimal parallel DONE implementation, however this master thesis has provided some proper tools to determine the derivative of this quest, making it easier for future researchers to determine their search directions.

Appendix A

Additional Results

A-1 Verification of the NFR Method

The results in Section 9-2 clearly show that the NFR method does not seem to work properly on optimization of the surrogate function (3-11). An explanation for this may simply be that the conjugate gradient framework does not work well with the RFE framework. It is however also a known problem that establishing new search directions from previous directions (6-17) may after a while result in a bad search direction due to numerical errors and since this bad search direction is then again used to establish a new search direction this is a diverging problem [6]. A way to solve this is to reset the search direction to the negative gradient every once in a while [6].

In order to ensure that the bad results are not due to wrong implementation, the same NFR implementation is used to optimize the quadratic test function

$$f_t(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T\boldsymbol{x} + \boldsymbol{x}^T\boldsymbol{\delta}, \tag{A-1}$$

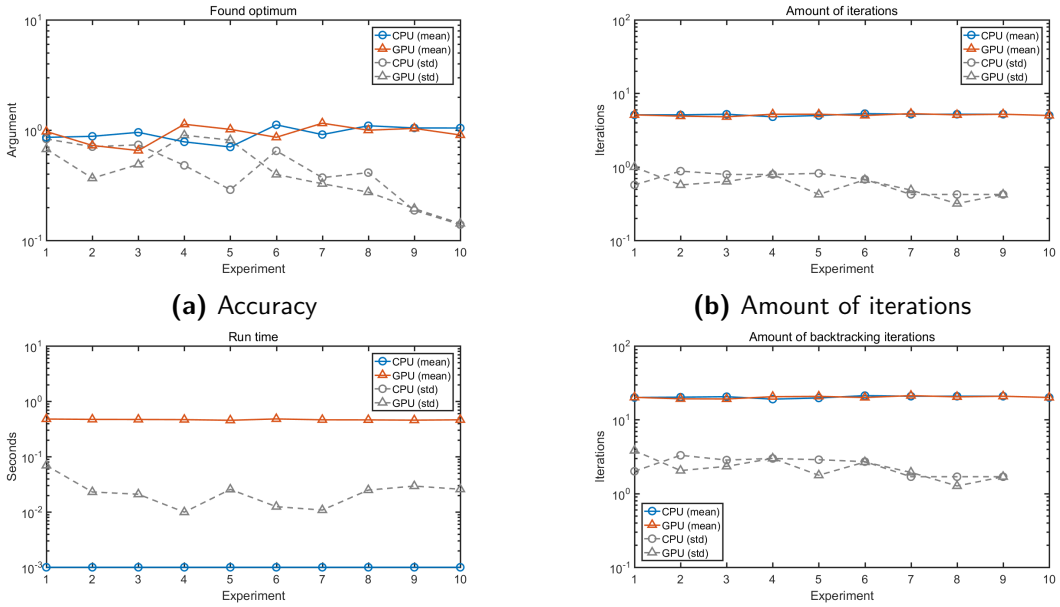
where the used parameters are chosen to be

$$\boldsymbol{\delta} \sim \mathcal{N}(\mathbf{0}_n, \boldsymbol{I}_n), \quad \boldsymbol{x}_0 \sim \mathcal{N}(\mathbf{0}_n, \boldsymbol{I}_n),$$

and the gradient is denoted by

$$\nabla_{\boldsymbol{x}} f_t(\boldsymbol{x}) = \boldsymbol{x} + \boldsymbol{\delta}. \tag{A-2}$$

Figure A-1 shows the results for this experiment. Due to time limitations, this experiment has been performed on set-up 2 from Table 8-3. As in Chapter 9, standard deviations that are below the lower boundary are not plotted. The same data preprocessing as in Appendix A-3 is applied here.



(a) Accuracy

(b) Amount of iterations

(c) Run time

(d) Amount of backtracking iterations

Figure A-1: Additional results for optimization using the NFR method. The solid lines represent the mean of ten performed experiments for each set of dimensions, the dashed lines represent the standard deviation. The horizontal axis represents the experiment number, see Table 8-1 for its relation to n.

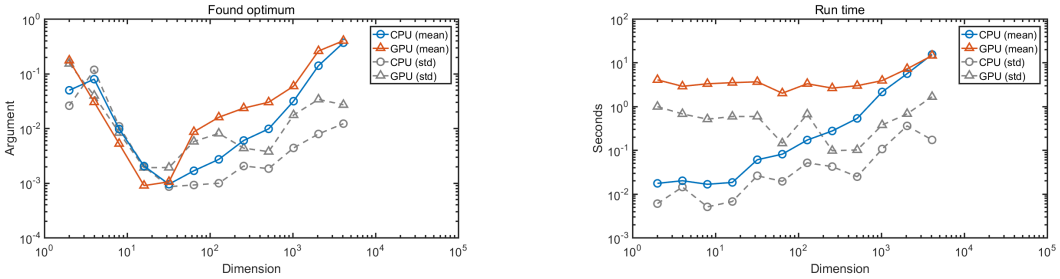
Before analysing the results it is important to note that the quadratic test function (A-1) is a less complicated function than the surrogate function (3-11) and hence the backtracking method is able to select very good step sizes. This explains the constant amounts of iterations and backtracking iterations and the constant run time for all tested dimensions. The most important result in this case is that the NFR implementation remains functional for all dimensions and that the accuracy does not decrease for increasing dimensions. This proves that the NFR implementation behaves as can be expected and is hence not wrongly implemented.

A-2 Choice of Optimization Method

From the results in Section 9-2 it turns out that both in the sequential and in the parallel case, the NGD method performs just as good as the BFGS method. This is a remarkable result, since the NGD method is a first-order optimization method and the BFGS method is a second-order optimization method. It does make sense however, since NGD is specifically designed to approach quasi-Newton convergence [31, 32, 34]. In order to make the right choice, both NGD and BFGS are implemented into a rudimentary parallel DONE implementation and both implementations are tested for accuracy and run time on the Gaussian experiment discussed in Section 8-3-3. Due to time limitations, these experiments have been performed on set-up 2 from Table 8-3.

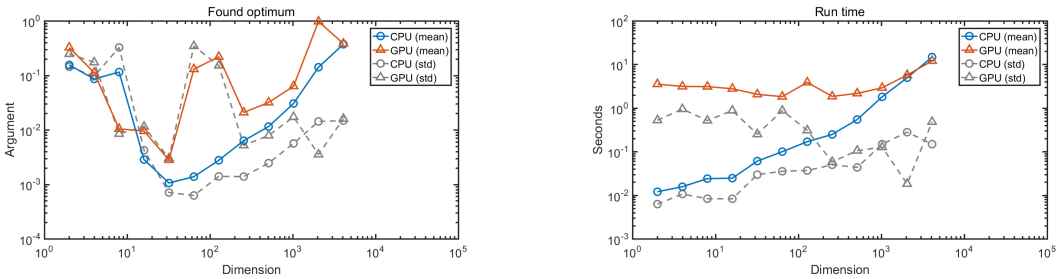
Figures A-2 and A-3 show the results of these tests. As can be seen with regard to run time both implementations perform similarly, however with regard to accuracy the implementation

containing the BFGS method performs better. Of course there may be another reason for this than NGD being worse than BFGS, for instance after some more extensive tuning both implementations may perform similarly with regard to accuracy as well. However since this currently is the only thing to base a decision on, BFGS appears to be the better choice.



(a) Accuracy **(b)** Run time

Figure A-2: Additional results for a DONE implementation using the BFGS method. The solid lines represent the mean of ten performed experiments for each set of dimensions, the dashed lines represent the standard deviation. The dimension on the horizontal axis represents m, see Table 8-1 for its relation to n and N.



(a) Accuracy **(b)** Run time

Figure A-3: Additional results for a DONE implementation using the NGD method. The solid lines represent the mean of ten performed experiments for each set of dimensions, the dashed lines represent the standard deviation. The dimension on the horizontal axis represents m, see Table 8-1 for its relation to n and N.

A-3 Data Preprocessing

Before the experiment results can be inspected in Chapter 9, several preprocessing steps need to be taken. Firstly it is important to realize that the lowest measurable run time value is one millisecond, hence any run time result that is zero should be set to one millisecond. This indicates that also for the standard deviation, a run time result below one millisecond is meaningless, so standard deviations below one millisecond are set to one millisecond as well. For all other criteria the lower limit is the smallest value that can be represented by a single-precision floating-point, however quick analysis of the results reveals that this lower limit is not reached. Subsequently there are several other reasons to declare an experiment result unfit for further analysis:

- A numerical error occurs which results in bad data.
- A practical error occurs, such as the GPU memory is full.
- The maximum amount of iterations is reached, hence no optimum is found.
- Divergence occurs, hence the results approach infinity.
- The found optimum is so far off that, though it may be a proper optimum, no fair comparison can be made with other results.

If a result is declared unfit based on one of these reasons, it is replaced by the mean of all other results in that experiment.

RFE fitting

In the LFR GPU results there is one case of divergence. Furthermore there are some cases of a full GPU memory. Since only the LFR, RLS and IQR methods make it through the full range of experiments, these are the only methods where this occurs. The dimensions listed in Table 8-1 for the final experiment are too large, causing the matrix $\mathbf{\Phi}_i \in \mathbb{R}^{m \times m}$ (or one of its inverse representations) to exceed the GPU memory, despite the fact that due to its symmetry only $\frac{1}{2}(m^2 + m)$ single-precision floating-points need to be stored.

Optimization

In the GD CPU results there are 25 cases where the maximum amount of iterations is reached, which mostly occurs at the highest dimensions. All results for the final experiment reach the maximum amount of iterations, which means that the final experiment is completely dismissed in this case. Remarkably, in the GD GPU results there are only four cases where the maximum amount of iterations is reached.

In the NGD CPU results there are two cases of divergence and one case of a bad optimum, which all three occur at the lowest dimensions. Likewise, in the NGD GPU there is one case of divergence, also at a low dimension.

In the NFR CPU results both divergence and reaching the maximum amount of iterations occur, causing all results for the second half of the experiments to be unfit for further analysis. In the results for the lower dimensions there are five cases of a bad optimum. Likewise, the NFR GPU results yields four cases of a bad optimum, and for the second half of the experiments no results can even be obtained.

In the BFGS CPU results there are two cases of numerical errors, one case of a bad optimum and one case where the maximum amount of iterations is reached. Similarly, in the BFGS GPU results there are two cases where the maximum amount of iterations is reached, one case of a numerical error and one case of a bad optimum.

The DONE algorithm

The experiments on the DONE algorithm itself do not yield any bad results, both for the inverted Gaussian experiments and the OBFN experiment.

Appendix B

CUDA Code

This page was intentionally left blank with regard to confidentiality regulations.

This page was intentionally left blank with regard to confidentiality regulations.

This page was intentionally left blank with regard to confidentiality regulations.

This page was intentionally left blank with regard to confidentiality regulations.

This page was intentionally left blank with regard to confidentiality regulations.

This page was intentionally left blank with regard to confidentiality regulations.

This page was intentionally left blank with regard to confidentiality regulations.

This page was intentionally left blank with regard to confidentiality regulations.

This page was intentionally left blank with regard to confidentiality regulations.

This page was intentionally left blank with regard to confidentiality regulations.

Appendix C

Convergence Rates

Definition of convergence rates

Linear convergence rates satisfy the condition

$$\lim_{k\to\infty} \frac{|f(\boldsymbol{x}_{k+1}) - f(\boldsymbol{x}^*)|}{|f(\boldsymbol{x}_k) - f(\boldsymbol{x}^*)|} = \mu, \tag{C-1}$$

where $\mu = 1$ indicates sublinear convergence, $0 < \mu < 1$ indicates linear convergence and $\mu = 0$ indicates superlinear convergence [6]. Quadratic convergence rates satisfy the condition

$$\lim_{k\to\infty} \frac{|f(\boldsymbol{x}_{k+1}) - f(\boldsymbol{x}^*)|}{|f(\boldsymbol{x}_k) - f(\boldsymbol{x}^*)|^2} = \mu, \tag{C-2}$$

where $0 < \mu < \infty$ [6].

Sublinear convergence rates

The GD method is stated to have convergence rate $\mathcal{O}(\frac{1}{k})$ [31, 32] and the NGD method is stated to have convergence rate $\mathcal{O}(\frac{1}{k^2})$ [31, 32, 34]. This indicates that the NGD method converges faster, however both convergence rates are still sublinear, as demonstrated by

$$\lim_{k\to\infty} \frac{\left|\frac{1}{k+1}\right|}{\left|\frac{1}{k}\right|} = \lim_{k\to\infty} \frac{k}{k+1} = \lim_{k\to\infty} \frac{-1}{k+1} + 1 = 1, \tag{C-3}$$

$$\lim_{k\to\infty} \frac{\left|\frac{1}{(k+1)^2}\right|}{\left|\frac{1}{k^2}\right|} = \lim_{k\to\infty} \frac{k^2}{k^2+2k+1} = \lim_{k\to\infty} \frac{-2k-1}{k^2+2k+1} + 1 = 1, \tag{C-4}$$

provided that $k > 0$.

Linear convergence rates

An example of linear convergence is $\mathcal{O}(\gamma^k)$, where $0 < \gamma < 1$, as demonstrated by

$$\lim_{k\to\infty} \frac{\left|\gamma^{k+1}\right|}{\left|\gamma^k\right|} = \lim_{k\to\infty} \gamma^{k+1-k} = \gamma, \tag{C-5}$$

provided that $k > 0$.

Superlinear convergence rates

An example of superlinear convergence is $\mathcal{O}(\gamma^{k^2})$, where $0 < \gamma < 1$, as demonstrated by

$$\lim_{k\to\infty} \frac{\left|\gamma^{(k+1)^2}\right|}{\left|\gamma^{k^2}\right|} = \lim_{k\to\infty} \frac{\gamma^{k^2+2k+a}}{\gamma^{k^2}} = \lim_{k\to\infty} \gamma^{2k+1} = 0, \tag{C-6}$$

provided that $k > 0$.

Quadratic convergence rates

An example of quadratic convergence is $\mathcal{O}(\gamma^{2^k})$, where $0 < \gamma < 1$, as demonstrated by

$$\lim_{k\to\infty} \frac{\left|\gamma^{2^{k+1}}\right|}{\left|\gamma^{2^k}\right|^2} = \lim_{k\to\infty} \frac{\gamma^{2^{k+1}}}{\gamma^{2\cdot 2^k}} = \lim_{k\to\infty} \gamma^{2^{k+1}-2^{k+1}} = 1, \tag{C-7}$$

provided that $k > 0$.

Appendix D

Interpretation of Nesterov's Method

Nesterov's accelerated gradient descent (NGD) method propagates not only the input vector $\boldsymbol{x}_k$ but also the momentum vector $\boldsymbol{v}_k$, which causes a different trajectory to be followed than the gradient descent trajectory. This trajectory may appear less efficient, however since momentum is taken into account as well this new trajectory may turn out to require less iterations. Take for instance a two-dimensional surrogate function $\hat{f}_i$, which can be considered as a curved surface embedded in a three-dimensional space. If a ball were to be dropped somewhere on this surface, it would roll towards one of the minima in the surface. If this ball is modelled as a point mass, the equation of motion becomes

$$M\ddot{\boldsymbol{x}} = -C\dot{\boldsymbol{x}} - \nabla_{\boldsymbol{x}}\hat{f}_i(\boldsymbol{x}), \tag{D-1}$$

where M is the mass, C is the friction coefficient, and the gradient $\nabla_{\boldsymbol{x}}\hat{f}_i(\boldsymbol{x})$ is considered to be a force acting on the point mass. The equation of motion can be converted to the state-space representation

$$\dot{\boldsymbol{x}} = M^{-1}\boldsymbol{v}, \tag{D-2}$$

$$\dot{\boldsymbol{v}} = -CM^{-1}\boldsymbol{v} - \nabla_{\boldsymbol{x}}\hat{f}_i(\boldsymbol{x}), \tag{D-3}$$

where the momentum vector $\boldsymbol{v}$ comes into play. By discretizing the obtained state-space representation using

$$\dot{\boldsymbol{x}} = \frac{\boldsymbol{x}_{k+1} - \boldsymbol{x}_k}{h}, \tag{D-4}$$

$$\dot{\boldsymbol{v}} = \frac{\boldsymbol{v}_{k+1} - \boldsymbol{v}_k}{h}, \tag{D-5}$$

where h is the sampling time, the final equations become

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + hM^{-1}\boldsymbol{v}_k, \tag{D-6}$$

$$\boldsymbol{v}_{k+1} = (1 - hCM^{-1})\boldsymbol{v}_k - h\nabla_{\boldsymbol{x}}\hat{f}_i(\boldsymbol{x}_k), \tag{D-7}$$

which turn out to be very similar to the equations (6-12, 6-13) used by the NGD method. Figures 6-4a and 6-4b show how the trajectory followed by the NGD method may appear less efficient but reaches the optimum within less iterations than the GD method does.

Bibliography

[1] L. Bliek, H. R. Verstraete, M. Verhaegen, and S. Wahls, "Online optimization with costly and noisy measurements using random fourier expansions," *arXiv preprint arXiv:1603.09620*, 2016.

[2] H. R. Verstraete, S. Wahls, J. Kalkman, and M. Verhaegen, "Model-based sensor-less wavefront aberration correction in optical coherence tomography," *Optics letters*, vol. 40, no. 24, pp. 5722–5725, 2015.

[3] L. Bliek, M. Verhaegen, and S. Wahls, "Data-driven minimization with random feature expansions for optical beam forming network tuning," *IFAC-PapersOnLine*, vol. 48, no. 25, pp. 166–171, 2015.

[4] L. M. Rios and N. V. Sahinidis, "Derivative-free optimization: a review of algorithms and comparison of software implementations," *Journal of Global Optimization*, vol. 56, no. 3, pp. 1247–1293, 2013.

[5] A. Rahimi and B. Recht, "Random features for large-scale kernel machines," in *Advances in neural information processing systems*, pp. 1177–1184, 2007.

[6] J. Nocedal and S. Wright, *Numerical optimization*. Springer Science & Business Media, 2006.

[7] J. Diaz, C. Munoz-Caro, and A. Nino, "A survey of parallel programming models and tools in the multi and many-core era," *IEEE Transactions on parallel and distributed systems*, vol. 23, no. 8, pp. 1369–1386, 2012.

[8] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[9] C. Vuik and C. Lemmens, *Programming on the GPU with CUDA (version 6.5)*. Delft Centre for Computational Science and Engineering, 2015.

[10] NVIDIA® Corporation, "Cuda c programming guide v7.5," 2015.

[11] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[12] G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew, "Extreme learning machine: theory and applications," *Neurocomputing*, vol. 70, no. 1, pp. 489–501, 2006.

[13] M. Nielsen, "Neural networks and deep learning." Source: www. neuralnetworksanddeeplearning. com, accessed: January 2016.

[14] A. H. Sayed and T. Kailath, "Recursive least-squares adaptive filters," *The Digital Signal Processing Handbook*, pp. 21.1–21.9, 1998.

[15] A. Rahimi and B. Recht, "Weighted sums of random kitchen sinks: Replacing minimization with randomization in learning," in *Advances in neural information processing systems*, pp. 1313–1320, 2009.

[16] J. A. Nelder and R. Mead, "A simplex method for function minimization," *The computer journal*, vol. 7, no. 4, pp. 308–313, 1965.

[17] M. J. Powell, "An efficient method for finding the minimum of a function of several variables without calculating derivatives," *The computer journal*, vol. 7, no. 2, pp. 155–162, 1964.

[18] M. J. Powell, "The newuoa software for unconstrained optimization without derivatives," in *Large-scale nonlinear optimization*, pp. 255–297, Springer, 2006.

[19] E. Brochu, V. M. Cora, and N. De Freitas, "A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning," *arXiv preprint arXiv:1012.2599*, 2010.

[20] A. Rahimi and B. Recht, "Uniform approximation of functions with random bases," in *Communication, Control, and Computing, 2008 46th Annual Allerton Conference on*, pp. 555–561, IEEE, 2008.

[21] B. W. Kernighan and D. M. Ritchie, "The c programming language," 2006.

[22] M. J. Flynn, "Very high-speed computing systems," *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1901–1909, 1966.

[23] J. M. Ortega, *Introduction to Parallel and Vector Solution of Linear Systems*. Springer Science & Business Media, 1988.

[24] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *IEEE micro*, no. 2, pp. 39–55, 2008.

[25] J. Nickolls and W. J. Dally, "The gpu computing era," *IEEE micro*, no. 2, pp. 56–69, 2010.

[26] NVIDIA® Corporation, "Cublas library v7.5," 2015.

[27] NVIDIA® Corporation, "Cusolver library v7.5," 2015.

[28] NVIDIA® Corporation, "Curand library v7.5," 2015.

[29] G. H. Golub and C. F. Van Loan, *Matrix computations*, vol. 3. JHU Press, 1996.

[30] L. Armijo, "Minimization of functions having lipschitz continuous first partial derivatives," *Pacific Journal of mathematics*, vol. 16, no. 1, pp. 1–3, 1966.

[31] S. Bubeck, "Orf523: Nesterov's accelerated gradient descent." Source: blogs. princeton. edu/ imabandit/ 2013/ 04/ 01/ acceleratedgradientdescent, accessed: April 2016.

[32] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *Proceedings of the 30th international conference on machine learning (ICML-13)*, pp. 1139–1147, 2013.

[33] W. Su, S. Boyd, and E. Candes, "A differential equation for modeling nesterov's accelerated gradient method: Theory and insights," in *Advances in Neural Information Processing Systems*, pp. 2510–2518, 2014.

[34] Y. Nesterov, "A method of solving a convex programming problem with convergence rate o (1/k2),"

[35] M. R. Hestenes and E. Stiefel, *Methods of conjugate gradients for solving linear systems*, vol. 49. NBS, 1952.

[36] J. Nocedal, "Updating quasi-newton matrices with limited storage," *Mathematics of computation*, vol. 35, no. 151, pp. 773–782, 1980.

Glossary

List of Acronyms

BLAS	basic linear algebra subprograms
BFGS	Broyden-Fletcher-Goldfarb-Shanno
CG	conjugate gradient
CPU	central processing unit
CS	coordinate search
CUDA	compute unified device architecture
D2H	device to host
DFP	Davidon-Fletcher-Powell
DONE	data-based online nonlinear extremum-seeker
FLOPS	floating-point operations per second
GD	gradient descent
GPU	graphics processing unit
H2D	host to device
IQR	inverse QR
LBFGS	limited memory Broyden-Fletcher-Goldfarb-Shanno
LFR	linear Fletcher-Reeves
LM	Levenberg-Marquardt
MIMD	multiple instruction stream, multiple data stream
MISD	multiple instruction stream, single data stream

N	Newton
NFR	nonlinear Fletcher-Reeves
NGD	Nesterov's accelerated gradient descent
NMS	Nelder-Mead simplex
OBFN	optical beam forming network
OCT	optical coherence tomography
ORR	optical ring resonator
PCD	Powell's conjugate directions
RFE	random Fourier expansion
RLS	recursive least-squares
RMS	root-mean-square
SIMD	single instruction stream, multiple data stream
SISD	single instruction stream, single data stream

List of Symbols

α	Step size, latency
β	Bandwidth
ϵ	RMS error
η	Parallel work fraction
γ	Support parameter
$\hat{f}$	Surrogate function
$\hat{J}$	Transformed least-squares cost function
∞	Infinity
λ	Regularization parameter
$\mathbb{R}$	Set of real numbers
$\mathbb{Z}$	Set of integers
$\mathcal{A}$	RFE domain
$\mathcal{N}$	Normal probability distribution
$\mathcal{O}$	Order of magnitude
$\mathcal{U}$	Uniform probability distribution
$\mathcal{X}$	Input domain
μ	Convergence rate factor
$\nabla \hat{f}$	Gradient of the surrogate function

$\nabla^2 \hat{f}$	Hessian of the surrogate function
$\nabla^2 J$	Hessian of the least-squares cost function
∇J	Gradient of the least-squares cost function
ν	Overhead factor
ϕ	Line search function
$\mathbf{0}_n$	Zero vector of size n
$\mathbf{1}_n$	Unit vector of size n
$\boldsymbol{\delta}$	Support vector
$\boldsymbol{\kappa}$	Input difference vector
$\boldsymbol{\Omega}$	Frequency matrix
$\boldsymbol{\omega}$	Frequency vector
$\boldsymbol{\Phi}$	Least-squares solution matrix
$\boldsymbol{\Theta}$	Orthogonal rotation matrix
$\boldsymbol{\theta}$	Gradient difference vector
$\boldsymbol{\xi}$	Exploration vector
$\boldsymbol{\zeta}$	Exploration vector
$\boldsymbol{A}$	Measurement matrix
$\boldsymbol{a}$	Measurement vector, support vector
$\boldsymbol{B}$	Inverse Hessian approximation
$\boldsymbol{b}$	Phase vector, support vector
$\boldsymbol{c}$	Support vector
$\boldsymbol{g}$	Least-squares support vector
$\boldsymbol{H}$	Hessian
$\boldsymbol{I}_n$	Identity matrix of size n by n
$\boldsymbol{P}$	Inverse least-squares solution matrix
$\boldsymbol{p}$	Search direction
$\boldsymbol{P}^{\frac{1}{2}}$	Square root inverse least-squares solution matrix
$\boldsymbol{r}$	Gradient
$\boldsymbol{s}$	Least-squares solution vector
$\boldsymbol{V}$	BFGS update matrix
$\boldsymbol{v}$	Momentum vector
$\boldsymbol{w}$	Weight vector
$\boldsymbol{x}$	Input vector
$\boldsymbol{x}^*$	Optimal input vector
$\boldsymbol{x}_{lb}$	Lower bound on the input vector
$\boldsymbol{x}_{ub}$	Upper bound on the input vector
$\boldsymbol{y}$	Objective function output vector
$\boldsymbol{z}$	Kernel vector
ψ	Single-layered neural network
ρ	Amount of parallel processors

σ	Standard deviation for normal probability distribution
τ	Support parameter
ε	Error margin
φ	Activation function
ϑ	Trust region model function
C	Friction coefficient
c_1	First Wolfe constant
c_2	Second Wolfe constant
D	Degree of parallelism
E	Efficiency
f	Objective function
f_t	Test function
G	Inverse Gaussian function
h	Sampling time
i	Measurement/cycle counter
J	Least-squares cost function
j	Counter
k	Iteration counter
M	Point mass
m	Amount of Fourier features
N	Amount of measurements/cycles
n	Input dimension
S	Speed-up vector
t	Computation time
t_{send}	Communication time
u	Amount of operations, amount of bits, amount of vectors
v_{send}	Communication speed
y	Objective function output

List of Operators

ceil()	Rounds its argument up to the nearest integer
diag()	Transforms a vector into a diagonal matrix
ker()	Computes a basis for the kernel of a matrix
max()	Returns the highest value of all of its arguments
min()	Returns the lowest value of all of its arguments