Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

A Numerical Approach to Nonlinear Regression Analysis by Evolving Parameters

Masterarbeit
im Arbeitsbereich Wissenschaftliches Rechnen, WR
Prof. Dr. Thomas Ludwig

Department Informatik
MIN-Fakultät
Universität Hamburg

vorgelegt von
Christopher Gerlach
am
29.06.2017

Gutachter: Prof. Dr. Thomas Ludwig
Dr. Michael Kuhn

Abstract

Nonlinear regression analysis is an important process of statistics and poses many challenges to the user. While linear models are analytically solvable, nonlinear models can in most cases only be solved numerically. What many numeric methods have in common, is that they require a proper starting point to reach satisfactory results. A poor choice of starting values can greatly reduce the convergence speed or in many cases even result in the algorithm not to converge at all. This thesis proposes a genetic numerical hybrid method to approach the problem from a nontraditional angle. The approach combines genetic algorithms with traditional numeric methods and proposes a design suitable for massive parallelization with GPGPU computing. It is shown that the approach can solve a large set of practical test problems without having to specify any starting values and that is fast enough for practical use, utilizing only consumer grade hardware.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This master's thesis is based on the bachelor's thesis *Numerical Evolution – Using Genetic Hybrid-Methods to Solve Nonlinear Regression Problems* by the same author[1].

The main goal of that thesis was to verify the validity of a hybrid approach to tackle the *starting value problem* of nonlinear regression analysis. The general idea of the proposed approach was to utilize *genetic algorithms* to optimize the starting value problem of traditional nonlinear regression methods.

By implementing a proof of concept, it was shown that the approach is viable in solving nonlinear regression problems of a specific form, consisting of up to 40 parameters. This thesis proposes a generalization of the approach, aiming to lift its restrictions, optimize it, and testing it on non-generated problems.

Because this work is a direct follow-up of another work, there are some overlaps in regards to the necessity to provide background and introductory information. This introduces the potential problem of *self-plagiarism* when both works are created by the same author. Self-plagiarism or *text recycling* is a topical issue and views on the acceptability of this practice are highly disputed. However, many available guidelines – such as the ones from the *Committee on Publication Ethics (COPE)*[2] – state it as unproblematic if the content in question is neither the main part of the work (e.g. republishing empirical data) nor presented in a misleading way.

In the spirit of academic honesty, the author would like to explicitly clarify that the following background chapters in this work are revised versions of similar chapters in the groundwork:

- 2.2.2 Nonlinear Regression
- 2.3 Genetic Algorithms

Additional references to the groundwork are treated in the same way as references to those of any other author.

1.1 Motivation

Nonlinear regression analysis is an important part of a statisticians toolbox. Although many regression problems in practice are linear – or can be converted to linear problems by applying transformations to the data – nonlinear dependencies are still very common and are even starting to take a more important role as complexity in data in many practical problems rises every day.

This poses several challenges to the statistician. Not only can nonlinear dependencies be modeled in countless ways, methods to solve nonlinear regression problems also require a lot of contextual knowledge and experience to be applied successfully.

An important aspect of nonlinear optimization problems is that they are generally not analytically solvable by nature. Contrary to linear ones, for which a closed solution is guaranteed to exist, nonlinear dependencies in most cases can only be extracted numerically. As with numerical methods in other mathematical fields, most nonlinear regression methods require a starting point to initiate the iterative process.

Determining good starting values is crucial to achieve satisfying results. Depending on how well the starting values are selected, a method might converge to a global extreme point, get stuck at a suboptimal solution, or will not even converge at all.

Transtrum et al. in their 2010 paper *Why are Nonlinear Fits to Data so Challenging?*[3] state:

> *"The estimation of model parameters from experimental data is astonishingly challenging. A nonlinear model with tens of parameters, fit (say) by least squares to experimental data, often demands weeks of human guidance to find a good starting point;"*

This problem gets even more severe when there are doubts about how to model the data. Because starting values may differ from model to model, they have to be determined again every time a new model is chosen. Transtrum et al. continue to stress that this can become a serious obstacle to progress, especially in situations where the statistician may want to automatically generate and explore a variety of models.

At the time of writing, the *starting value problem* or *initial guess problem* remains to be an ongoing issue. Although there have been several advancements over the past decades to mitigate the problem, there currently is no *silver bullet* solution that leads to satisfying results in every situation.

This master's thesis proposes a nontraditional approach to the starting value problem that is able solve a wide variety of practical nonlinear regression problems.

1.2 Outline

This thesis is organized in the following chapters:

1. Introduction
2. Background
3. Related Work
4. The Parameter Evolution Approach
5. Implementation
6. Evaluation
7. Conclusion
8. Outlook

After the introduction, chapter 2 provides the fundamental concepts and definitions this thesis is based on. It includes a brief summary of some concepts of linear algebra, and an introduction to nonlinear regression analysis and genetic algorithms. Because one of the goals of this thesis is to examine the benefits of utilizing massive parallelism for the approach, the background chapter also includes an overview of current GPGPU technologies.

Before the approach is introduced, chapter 3 formally defines the starting value problem and shows how it was traditionally approached in the past. Additionally, the chapter also provides an overview of the Hybrid Approach, on which the approach of this thesis is based on.

Chapter 4 then proposes a nontraditional approach to the starting value problem. This is done by first analyzing the disadvantages of the methods presented in the previous chapter and then defining a set of requirements an ideal solution should comply to. Based on these requirements, the chapter continues by formally defining the goals of this thesis, which include a set of properties the approach should hold. The remainder of the chapter presents a detailed description of the approach and each of its individual parts.

After its formal description, chapter 5 presents an overview of an exemplary implementation of the approach. While chapter 4 describes the approach on an abstract level, this chapter discusses the less obvious details and challenges of actually implementing the approach for practical use. Specifically, a significant chunk of the chapter shows how the approach can utilize massive parallelism with the usage of C++ AMP.

To assess whether the approach actually satisfies the goals that are defined in chapter 4, the exemplary implementation is used to analyze its properties. The results of that evaluation, along with a description of the setup, are presented in chapter 6.

The thesis concludes by assessing whether the goals of the thesis were actually met. It is also briefly discussed how the approach could be improved and how it could be used in practice.

Chapter 2

Background

The previous chapter provided a brief description of the problem the approach proposed in this thesis is trying to solve and the motivation to do so.

The next step would be to formally define the starting value problem and to take a look at related work and examine how it is typically handled in practice. However, to make the thesis also accessible to readers who are less familiar with regression analysis and genetic algorithms, it is necessary to first take a look at the required background information relevant to the topics covered in this thesis.

The first section covers some basic topics in the mathematical subfield of linear algebra, while the second section provides a basic introduction to regression analysis in general and nonlinear regression in particular. Afterwards, the chapter concludes with an introduction to genetic algorithms as well as GPGPU computing, which are both being used in the approach proposed in this thesis.

2.1 Linear Algebra

Although regression analysis is a part of statistics, its mathematical descriptions are largely based on concepts from the field of *linear algebra (LA)*. The following section provides a brief summary of those concepts that are essential to the understanding of nonlinear regression analysis. The goal is to not only introduce some maybe lesser known concepts to the reader, but also to describe the notation and terminology used in this thesis. Note that this summary still assumes some basic prior understanding of linear algebra and is probably not sufficient for people who are not familiar with it at all. If the reader does not feel comfortable after reading what is presented here, there is a wide variety of literature that provides a proper introduction to the topic.

Most definitions in this chapter are taken from the *Handbook of Linear Algebra*[4].

2.1.1 Real Numbers Assumption

Before starting to define anything, it should be emphasized that all mathematical definitions in this thesis are restricted in a mathematical sense. As with other mathematical areas, many ideas from linear algebra can be defined for arbitrary algebraic structures with certain properties. To be as general as possible, many textbooks define concepts from linear algebra either for arbitrary fields or for the field of complex numbers $\mathbb{C}$.

However, because regression analysis is usually applied to real valued data, all mathematical definitions used in this thesis will be defined with the assumption of the field of real numbers $\mathbb{R}$.

2.1.2 Notation and Terminology

A common hindrance in communicating mathematical concepts is that notations and terminology are not always universal. This is especially true in linear algebra, where authors may use very different conventions when referring to vectors, matrices or other concepts. For this reason, this section will provide a brief summary of the fundamental components of linear algebra, along with their corresponding notation.

Matrices and Vectors

This thesis uses the following notation for vectors and matrices:

- Vectors are denoted as bold, lower-case letters (e.g $\boldsymbol{v}$)
- Vector entries are denoted as indexed lower-case letters (e.g. v_i)
- Matrices are denoted as bold, upper-case letters (e.g. $\boldsymbol{A}$)
- Matrix entries are denoted as double indexed lower-case letters (e.g. a_{ij})

Identity Matrix

The *identity matrix* $\boldsymbol{I}$ is the *neutral element* of the matrix multiplication:

$$\boldsymbol{A} \cdot \boldsymbol{I} = \boldsymbol{A}$$

Note that in this thesis, $\boldsymbol{I}$ is used exclusively for the identity matrix and for nothing else.

Inverse Matrix

The *inverse matrix* is the *inverse element* of the matrix multiplication:

$$\boldsymbol{A} \cdot \boldsymbol{A}^{-1} = \boldsymbol{A}^{-1} \cdot \boldsymbol{A} = \boldsymbol{I}$$

Diagonal Matrix

A matrix $\boldsymbol{A}$ is *diagonal*, if $a_{ij} = 0$ for every $i \neq j$.

For example:

$$\boldsymbol{A} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$

Unitary Matrix

A matrix is *unitary*, if its transpose is also its inverse:

$$\boldsymbol{A} \cdot \boldsymbol{A}^{\intercal} = \boldsymbol{A}^{\intercal} \cdot \boldsymbol{A} = \boldsymbol{I}$$

Hermitian Matrix

A matrix is *hermitian*, if it is equal to its own transpose:

$$\boldsymbol{A} = \boldsymbol{A}^{\intercal}$$

2.1.3 Singular Value Decomposition

In linear algebra, a *matrix decomposition* or *matrix factorization* is the representation of a matrix as a product of matrices. These matrices often exhibit certain properties that are beneficial to a particular set of problems. There are many different types of matrix decompositions, each with their individual areas of application. The theorem of the SVD states that a matrix $\boldsymbol{A}$ can be factorized into a product of three matrices:

$$\boldsymbol{A} = \boldsymbol{U} \cdot \boldsymbol{\Sigma} \cdot \boldsymbol{V}^{\intercal},$$

where:

- $\boldsymbol{U}$ is a $m \times m$ unitary matrix
- $\boldsymbol{\Sigma}$ is a $m \times n$ positive diagonal matrix
- $\boldsymbol{V}^{\intercal}$ is the transpose of a $n \times n$ unitary matrix $\boldsymbol{V}$

There are many different algorithms for calculating the SVD, which differ greatly in terms of efficiency and numerical stability.

2.1.4 Moore-Penrose Pseudoinverse

A pseudoinverse of a matrix is a generalization of the inverse matrix. A generalized inverse has some properties of the inverse, but not necessarily all of them. Such a matrix can then be used in situations where an inverse is needed, but may be difficult to determine.

Its most widely known representative is the *Moore-Penrose pseudoinverse*, which was independently described by E.H. Moore in 1920, Arne Bjerhammar in 1951 and Roger Penrose in 1955. In fact, the Moore-Penrose pseudoinverse is so prevalent in this context, that it is often used synonymously with the term pseudoinverse, without any further specification. This practice is also be applied here.

A pseudoinverse $\boldsymbol{A}^+$ of a matrix $\boldsymbol{A}$ is defined as a matrix satisfying all of the following criteria:

1. $\boldsymbol{A}\boldsymbol{A}^+\boldsymbol{A} = \boldsymbol{A}$
2. $\boldsymbol{A}^+\boldsymbol{A}\boldsymbol{A}^+ = \boldsymbol{A}^+$
3. $(\boldsymbol{A}\boldsymbol{A}^+)^\intercal = \boldsymbol{A}\boldsymbol{A}^+$
4. $(\boldsymbol{A}^+\boldsymbol{A})^\intercal = \boldsymbol{A}^+\boldsymbol{A}$

The first property specifies that $\boldsymbol{A}\boldsymbol{A}^+$ does not need to be a general identity matrix, but has to map all its column vectors to itself. The second property necessitates $\boldsymbol{A}^+$ to be a weak inverse of the multiplicative semigroup, while the third and fourth property requires $\boldsymbol{A}\boldsymbol{A}^+$ as well as $\boldsymbol{A}^+\boldsymbol{A}$ to be hemertian.

The Moore-Penrose Pseudoinverse has many areas of application, one of which is the solution of systems of linear equations.

2.2 Regression Analysis

In statistics, *regression analysis (RA)* describes a process to estimate the relationship among variables. The goal is to determine the dependencies between a *dependent variable* or *response variable* and one or more *independent variables* or *predictor variables* in regards to a specific mathematical model. It has a wide range of applications, but is mostly used for prediction and forecasting[5].

A simple example from economics is that of a used car salesman, who is interested in the question of the expected sales price of a particular vehicle – the response variable – dependent on factors such as the cars mileage, its age and the number of previous owners – the predictor variables. Using regression analysis, this relationship can be modeled based on records of past sales. A response and its corresponding predictors are also referred to as a *data point*.

There exist a lot of different regression techniques, all with their own advantages, disadvantages and areas of use. What follows is a brief introduction to the subfield of nonlinear regression analysis and the notation and terminology used in this thesis. For a more deeper look into the techniques presented here, the author would like to refer the readers to the wide variety of literature about this topic, such as [6] or [7].

2.2.1 Notation And Terminology

Regardless which specific method is used, a general regression model for a single data point can be written as

$$y = f(\boldsymbol{x}, \boldsymbol{\theta}) + Z,$$

where f is the *model function* with (unknown) parameters $\boldsymbol{\theta}$ and y is the response variable for a vector $\boldsymbol{x}$ of associated predictor variables. Additionally, Z denotes a random variable that is used to model potential uncertainty in the data points, which is often caused by measurement errors and noise.

For a set of data points, this definition can be extended to

$$\boldsymbol{y} = \hat{f}(\boldsymbol{X}, \boldsymbol{\theta}) + \boldsymbol{Z}, \tag{2.1}$$

where $\hat{f}$ is the *vector-valued* model function and $\boldsymbol{y}$ is the vector of all response variables for the matrix $\boldsymbol{X}$ of all associated predictor variables.

The input matrix $\boldsymbol{X}$ – also sometimes referred to as *regressor* – is a $K \times N$ matrix of predictor variables,

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{x}_0 \\ \vdots \\ \boldsymbol{x}_k \end{bmatrix} = \begin{bmatrix} x_{0,0} & \cdots & x_{0,N} \\ \vdots & \ddots & \vdots \\ x_{K,0} & \cdots & x_{K,N} \end{bmatrix},$$

where K denotes the number of data points and N refers to the number of predictor variables.

The goal of regression analysis is to determine the unknown parameters $\boldsymbol{\theta}$, such that 2.1 is approximated as close as possible.

2.2.2 Nonlinear Regression

One particular challenge of regression analysis is to determine the form of the model function f. In some areas of application, such as physics, chemistry or biology, the form can often be determined by theoretical considerations[5]. However,

this is not always the case, especially in applications of regression analysis in which the measured data is not based on a theoretical model. In those cases, the form of the model has to be determined by manually examining the data, which requires a lot of experience and intuition from the experimenter and gets increasingly difficult the more variables and parameters are present in the regression. Regardless of what particular method is used, the general form of the model can be divided into two categories: *linear* and *nonlinear* models.

Looking back at the used-car example above, one feasible model could be

$$f(\boldsymbol{x}_k, \boldsymbol{\theta}) = \theta_1 \cdot x_1 + \theta_2 \cdot x_2 + \theta_3 \cdot x_3.$$

This model is linear, because it is linear in all three parameters θ_j. Another possible model could be

$$f(\boldsymbol{x}_k, \boldsymbol{\theta}) = e^{\theta_1 \cdot x_1 + \theta_2 \cdot x_2 + \theta_3 \cdot x_3}.$$

This model is nonlinear in all parameters. However, using a process called *data transformation*[8], the regression function can be converted into the linear model

$$f^t(\boldsymbol{x}_k, \boldsymbol{\theta}) = \log y = \theta_1 \cdot x_1 + \theta_2 \cdot x_2 + \theta_3 \cdot x_3.$$

Data transformations make the family of linear models very flexible and are often used avoid nonlinear dependencies when possible. Transformation of the data might also have other benefits, like increasing the stability of regression methods. However, it should be noted that data transformations can get quite complex and that sometimes transformation can decrease the quality the results. Some transformations change the model function in such a way, that strictly speaking another problem is optimized, whose minima do not necessarily have to coincide with those of the original problem.

In addition to transformations, the x_j's in a linear model can also include squares, cross products, higher powers and more, without changing the linearity of the model. The important requirement is that the expression must be linear in the parameters.

Transforming a nonlinear model into a linear model is not always possible. The model function f could also look like

$$f(\boldsymbol{x}_k, \boldsymbol{\theta}) = x_1^{\theta_1} + \theta_2^{x_2} + \theta_3 \cdot x_3.$$

Here, f is nonlinear in the parameters θ_1 and θ_2. Because it is not possible to linearize this model by means of suitable transformations, it can also be described

as *intrinsically nonlinear*.

Nonlinear dependencies tend to be used when they are either suggested by theoretical considerations or to build known nonlinear behavior into a model. And even when a linear approximation might work well, a nonlinear model may still be used to retain a clear interpretation of the parameters.

Regression problems with strictly nonlinear dependencies fall into the domain of *nonlinear regression analysis (NRA)*.

Method of Nonlinear Least Squares

To estimate the parameter vector $\boldsymbol{\theta}$, the research of regression analysis produced a wide variety of different procedures over the past few decades. These methods differ in their effectiveness and efficiency. One of the simplest and most commonly used estimators is the method of *nonlinear least squares (NLS)*[9]. This method finds its optimum when the sum of squared residuals S (from here on referred to as the *sum of squares (SoS)* or *diffsum*)

$$S(\boldsymbol{\theta}) = \sum_{k=1}^{K} r_k^2. \tag{2.2}$$

is a minimum. Each residual corresponds to one of K sets of (measured) data points. A residual r_k is hereby defined as the difference between the actual values of the response variable y_k and the value predicted by the model function f, with respect to the current guess of parameters $\boldsymbol{\theta}$

$$r_k = y_k - f(\boldsymbol{x}_k, \boldsymbol{\theta}).$$

This way, the method of least squares turns the optimization problem of f into a minimization problem of S. The minimum value of S occurs when its first derivative S' is zero. Since non-trivial models contain more than one parameter, the derivative is actually the gradient of S

$$S(\boldsymbol{\theta})' = \left(\sum_{i=1}^{n} r_i^2 \right)' = \frac{\partial S}{\partial \boldsymbol{\theta}} = 2 \sum_i r_i \cdot \frac{\partial r_i}{\partial \boldsymbol{\theta}} = 0 \quad (j = 1, \ldots, n).$$

In a nonlinear system, the derivatives $\frac{\partial S}{\partial \boldsymbol{\theta}}$ are functions of both the predictor variables and the parameters and therefore do not have an analytical or closed solution. Instead, initial values must be chosen for the parameters. Then, the parameters are refined iteratively, that is, the values are obtained by successive approximation.

Applying a first order Taylor expansion to S yields

$$\widehat{S}(\boldsymbol{\theta}) = S(\boldsymbol{\theta}^h) + \boldsymbol{\nabla}^\intercal S(\boldsymbol{\theta}^h) \cdot (\boldsymbol{\theta} - \boldsymbol{\theta}^h) + \frac{1}{2}\boldsymbol{\nabla}^2 S(\boldsymbol{\theta}^h) \cdot (\boldsymbol{\theta} - \boldsymbol{\theta}^h)^\intercal.$$

This leads to the update rule

$$\boldsymbol{\theta}^h - \boldsymbol{\theta}^{h+1} = \delta\boldsymbol{\theta} = -\boldsymbol{\nabla}^2 S(\boldsymbol{\theta}^h)^{-1} \cdot \boldsymbol{\nabla} S(\boldsymbol{\theta}^h). \tag{2.3}$$

This rule is the multidimensional case of the popular *Newton method* as it is used in optimization, and constitutes the basis for many more refined methods like the Gauss-Newton and Levenberg-Marquardt algorithms.

2.2.3 Gauss-Newton Algorithm

Although the update rule (2.3) already provides a valid method to minimize the nonlinear least squares problem, it is difficult to compute due to the inverse of the Hessian $\boldsymbol{\nabla}^2 S(\boldsymbol{\theta}^h)^{-1}$. Using the so called *Gauss-Newton* or *small-residual* approximation, the Hessian can also be written like[10]

$$\begin{aligned}
\boldsymbol{\nabla}^2 S(\boldsymbol{\theta}^h) &= \sum_m \left(\frac{\partial r_m}{\partial \theta_\mu} \cdot \frac{\partial r_m}{\partial \theta_\nu} + r_m \cdot \frac{\partial^2 r_m}{\partial \theta_\mu \partial \theta_n u} \right) \\
&\approx \sum_m \frac{\partial r_m}{\partial \theta_\mu} \cdot \frac{\partial r_m}{\partial \theta_\nu} \\
&= (\boldsymbol{J}^\intercal \boldsymbol{J})_{\mu\nu},
\end{aligned}$$

where $\boldsymbol{J}$ denotes the Jacobian matrix of the residual vector $\boldsymbol{r}$. Inserting this into (2.3) and additionally using $\boldsymbol{\nabla} S(\boldsymbol{\theta}^h) = \boldsymbol{J}^\intercal \cdot \boldsymbol{r}$ yields:

$$\delta\boldsymbol{\theta} = (\boldsymbol{J}^\intercal \boldsymbol{J})^{-1} \cdot \boldsymbol{J}^\intercal \cdot \boldsymbol{r} \tag{2.4}$$

$$\Leftrightarrow (\boldsymbol{J}^\intercal \boldsymbol{J}) \cdot \delta\boldsymbol{\theta} = \boldsymbol{J}^\intercal \cdot \boldsymbol{r} \tag{2.5}$$

Because $\boldsymbol{J}^\intercal \cdot \boldsymbol{J}$ is symmetrical and positive definite, a decomposition method like a Cholesky, QR or SVD decomposition can be used to calculate $\delta\boldsymbol{\theta}$[11].

This leads to the Gauss-Newton algorithm, which involves the following steps:

1. Guess appropriate starting parameters for $\boldsymbol{\theta}$
2. Compute the current values for the Jacobian $\boldsymbol{J}$ and the residual vector $\boldsymbol{r}$
3. Perform an update as directed by (2.5)
4. Repeat steps (2) to (3) until the algorithm meets one of the defined stopping rules

The Gauss-Newton algorithm is often used as a basis for more advanced methods, such as the popular Levenberg-Marquardt algorithm, which is discussed in more detail in chapter 4.

2.3 Genetic Algorithms

The term *genetic algorithm (GA)* describes a subclass of the so called *evolutionary algorithms (EAs)*, which include a number of metaheuristic methods to solve optimization problems. Evolutionary algorithms are inspired by the process of biological evolution, such as reproduction, mutation, recombination and selection. They include, among others, the fields of genetic and evolutionary programming, gene expression programming, neuroevolution and genetic algorithms. Genetic algorithms in particular mimic the process of natural selection. Although a wide range of EAs have been around since the 1950's, John H. Holland is often credited for the popularization of GAs in the late 1980's and early 1990's[12].

Although actual implementations of GAs can differ on a case by case basis, the so called *genetic process (GP)* generally consist of the following components:

1. Generate a random starting *population* P of genetic *chromosomes* C_i
2. Evaluate and calculate the fitness of each chromosome $C_1, ..., C_n$ out of P
3. Use a selection method to select chromosomes out of P
4. Use the crossover and mutation operators to create a new generation $\widehat{P}$ of chromosomes $\widehat{C}_1, ...\widehat{C}_n$
5. Set $P = \widehat{P}$
6. Repeat (2) to (5) until the desired result or a stopping rule is reached

Each chromosome represents an encoded solution to a particular optimization problem. The chromosomes are only decoded in step (2), when the fitness for each chromosome is set. Because the chromosomes that are used to build the next population are selected based on their fitness, each generation should on average be fitter than the last one. This procedure is repeated until an appropriate result is found or a stopping rule is met. The algorithm usually terminates when a predefined number of generations were produced.

A simple example to demonstrate the basic principle behind genetic algorithms is to find the square root of an integer number. Although analytical solutions for this problem exist – and are undeniably more efficient – it is an inverse problem, where calculating the square is much simpler than calculating the square root. In this example, the fitness of each chromosome can be determined by calculating the square of the encoded number and then determining the difference to the number the square root is attempted to be found for. For example, when the looking for the square root of 25, the fitness could be calculated as follows:

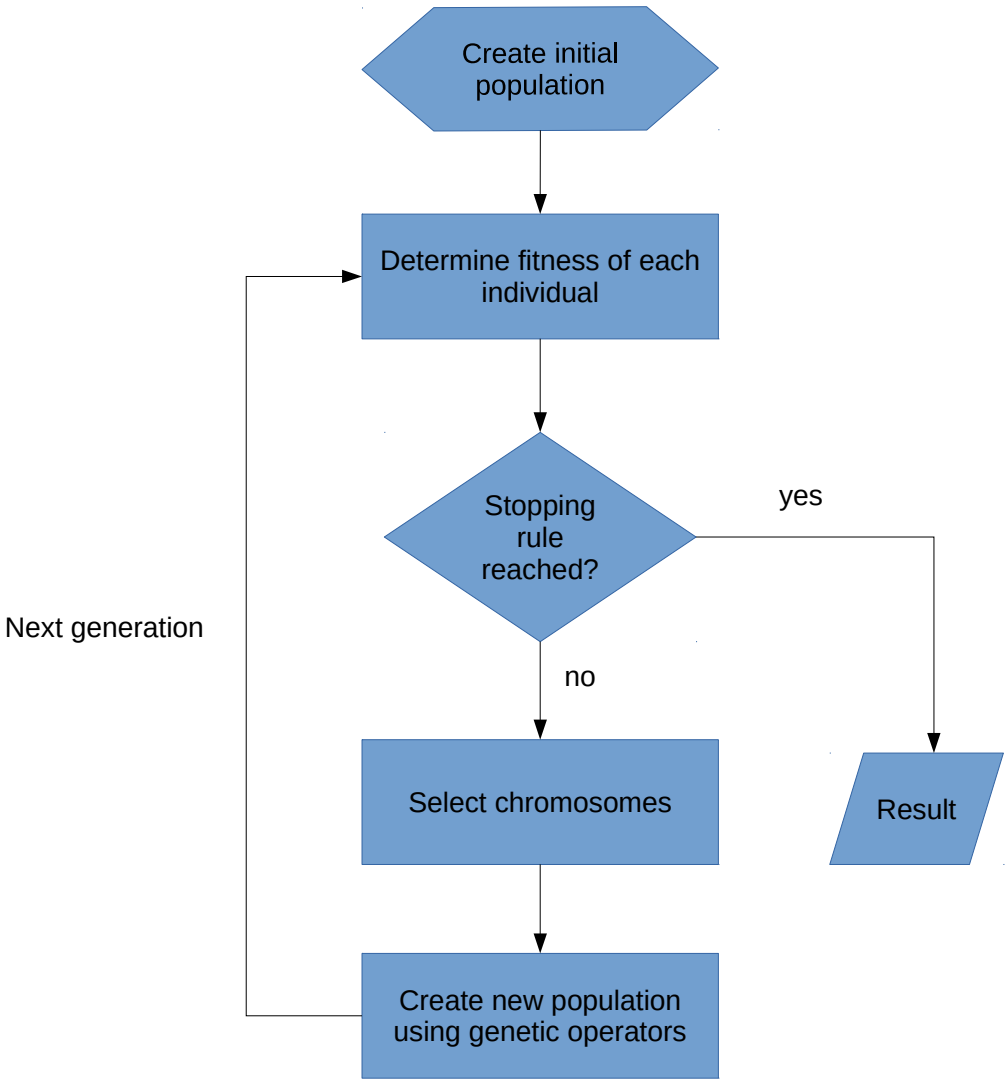

Figure 2.1: Workflow of the genetic process.

$$\begin{aligned} &\text{decode}(C_1) = 2 \\ &\text{decode}(C_2) = 7 \end{aligned}$$

$$\begin{aligned} &\Rightarrow \text{fitness}(C_1) = |2 \cdot 2 - 25| = |4 - 25| = 21 \\ &\Rightarrow \text{fitness}(C_1) = |7 \cdot 7 - 25| = |49 - 25| = 24 \end{aligned}$$

Note that without negation, C_2 would have a higher fitness value than C_1, although it is farther away from the optimal solution. This is a common issue when the

optimization problem consists of minimizing a difference and can be rectified by negating the fitness values, before assigning them to the chromosomes.

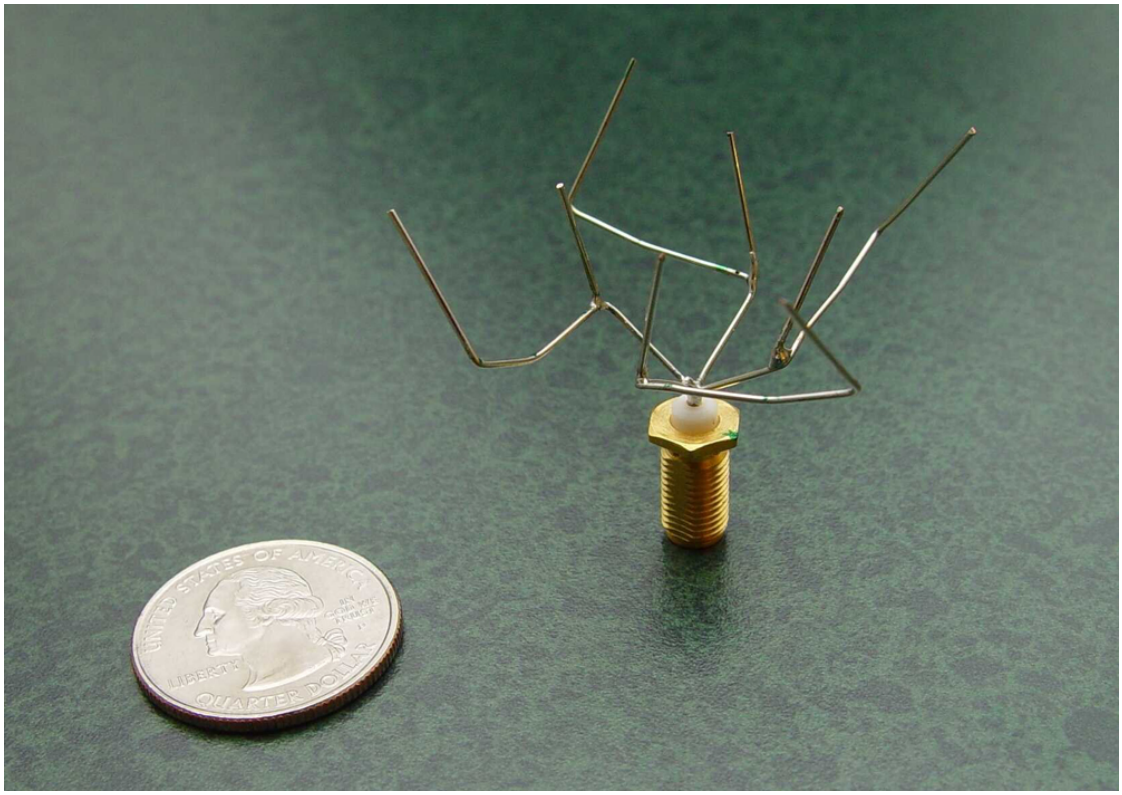

Figure 2.2: X-band antenna of the NASA ST5 spacecraft. The shape was found by genetic algorithms to create the best radiation pattern.

One advantage of genetic algorithms is that they only operate on the raw encoded data, which makes them independent of an analytical solution of the problem they are optimizing. The only requirement is that each encoded result must be assessable in a gradual fashion. Intermediate results that differ in quality also have to differ in their fitness values. For example, problems that have a big solution domain, but whose results can only be evaluated to be correct or false, are unsuitable for genetic algorithms.

Because the usage of GAs potentially involves a lot of solution evaluations, each evaluation should also be considerably low in computational cost. For this reason, GAs are often used for inverse problems, which are easy to verify, but where analytical solutions are either exponentially expensive or do not exist at all[13]. Figure 2.2 shows a popular and recent example, where genetic algorithms were used to find the best radiation pattern of a X-band antenna used by the NASA within the Space Technology 5 mission in 2006.

The genetic process, including each of its individual components, is discussed in more detail in chapter 4.

2.4 General-Purpose GPU Computing

The term *General-purpose GPU computing (GPGPU)* refers to the usage a *graphics processing unit (GPU)* to perform computation that is traditionally done on a CPU. A good introduction to the topic is given in the book *C++ AMP: Accelerated Massive Parallelism with Microsoft Visual C++*[14] by Kate Gregory and Ade Miller.

2.4.1 History

The idea of GPGPU began to attract popular interest when it became clear to many people that the so called *free lunch* had ended. The term refers to the expansive growth of hardware development that started with the introduction of the so called *personal computer* around 1975. For around 30 years computers not only became more popular, but also cheaper and fast every year; with every chip having higher clock speeds, more cache and better performance than the generation before it. This meant that slow software was not necessarily such a big problem, because already six months later, new CPUs could potentially be so much faster to make performance a non-issue.

Around the year 2005, manufactures began to encounter physical limitations in their CPU designs. Although they were still able to increase the number of transistors, issues such as proper dissipation of the heat from the chip resulted in much lower differences in clock speeds between generations compared to the years before. At this point in time, microprocessor companies such as Intel began to introduce the concept of *multicore* CPUs, which aimed to increase the overall performance by distributing the overall load to individual computation units.

Although multiple cores increase the performance of a multithreaded system by asynchronously processing multiple tasks at the same time, they rarely load a CPU to even more than 50%. To use the full potential of a multi-core CPU, the software has to be *parallel-aware*, which means that is has to include algorithms that can be efficiently parallelized. This is rarely the case for most conventional software.

On the other hand, software that is parallel-aware often times almost scales directly with the number of cores, meaning that doubling the number of cores would also roughly double the performance. However, even years later, CPUs with more than four cores were very rare and usually restricted to data centers. At the same time, hardware vendors began to develop GPUs with dozens, hundreds and now even thousands of cores. Although these cores are very different from CPU cores and are not suited for asynchronous computing, they allow the utilization of massive parallelism when used properly. This lead to the concept of *heterogeneous computing (HC)*, which refers to the general idea of using different types of processors together and distributing the workload to the one that is best suited for the corresponding type of computation. The term GPGPU computing refers to

the combination of a CPU and a GPU.

2.4.2 Technologies

Very early on, GPGPU computing was only possible by using APIs designed for graphics rendering. But the fast-growing interest in the topic quickly lead to the development of several technologies over the past 10 years that are specifically designed for HC in general and GPGPU computing in particular. What follows is a short overview of the most popular developments, together with a brief description of their advantages and disadvantages.

OpenCL

The *Open Computing Language (OpenCL)* is an open C-like programming language based on the *Open Graphics Library (OpenGL)* and developed by the Khronos Group[15]. Contrary to other technologies, OpenCL imposes no restrictions on the specific graphics cards or hardware, and is available on a variety of platforms. This means that an OpenCL program is not bound to be used on a GPU and can also be parallelized on other suitable hardware. As with OpenGL, many (free) tools are available to write, test and debug OpenCL applications. The main disadvantage is that because it is cross-platform, cross-hardware and even cross-language, it is not as easy to use as other solutions.

CUDA

The *Compute Device Unified Architecture (CUDA)*[16] refers to a GPGPU-specific programming language developed by Nvidia, who also manufacture CUDA-specfic hardware. The full name of the language is *CUDA C*, which like OpenCL also is a C-like language. However, compared to OpenCL, CUDA C provides higher-level abstractions and in recent years has moved closer to C++ rather than C. The main advantage of CUDA is that the language allows simpler GPU invocation syntax and also to share code between the CPU and the GPU. CUDA is under active development still continues to gain new features and libraries. The main disadvantages of CUDA is its restriction to Nvidia hardware.

Direct3D

The term *Direct3D* refers to a number of technologies for graphics programming for Windows[17]. One part of Direct3D is the *DirectCompute* API, which specifically supports GPGPU computing. DirectCompute is based on the *High Level Shader Language (HLSL)*, which is mostly used in game development. The main disadvantage of DirectCompute is that it is restricted to being used on Windows.

C++ AMP

The C++ *Accelerated Massive Parallelism (AMP)*[18] is an open specification by Microsoft that describes a native programming model for data parallelism. Although Microsoft's own implementation is based on DirectX 11 and is therefore limited to Windows, the standard being open means that C++ AMP can theoretically be implemented on any platform. For example, in 2014, AMD released an implementation for Linux.

Aside from the already mentioned disadvantages, all of the aforementioned technologies require a programmer to not only learn a new API, but also a new programming language. Additionally, OpenCL and DirectCompute lack C++ abstractions such as type safety and genericity.

C++ AMP on the other hand consists mostly of library code and only a small extension of the C++ language. Although C++ AMP code cannot use the full feature set of C++, it is fully compatible with it, while still being able to use many high-level concepts, such as genericity, type-safety, classes, function overloading or lambda functions.

SYCL

The *C++ Single-source Heterogeneous Programming for OpenCL (SYCL)*[19] is an open specification for a cross-platform abstraction layer for OpenCL, developed by the Khronos Group. Its main goal is to allow full usage of OpenCL, using only standard C++. It would mitigate many of the mentioned disadvantages of OpenCL and could potentially also completely supersede C++ AMP as a technology that provides high-level access to massive parallelism. Unfortunately, at the point in time this thesis was started, no implementation of SYCL was available publicly available.

Chapter 3

Related Work

The previous chapter provided an overview of the general concepts this thesis is based on. But before introducing the approach of this work, it is important to examine how the starting value problem was tackled in the past.

This chapter is divided into three parts. After the starting value problem is formally defined, the second section provides a brief overview of traditional methods and strategies that are currently used or were used in the past to solve the same problem. Then, the third section shows how the Hybrid Approach attempted to tackle the problem from a non-traditional angle.

3.1 The Starting Value Problem

Although it should be clear to most readers by now what the starting value problem is, it is formally defined here to avoid any confusion.

In nonlinear regression analysis, the *starting value problem* refers to the fact that most nonlinear regression methods have to be initialized with a set of starting values for the regression parameters. For example, recalling 2.2.3, the Gauss-Newton algorithm was described as follows:

1. Guess appropriate starting parameters for $\boldsymbol{\theta}$
2. Compute the current values for the Jacobian $\boldsymbol{J}$ and the residual vector $\boldsymbol{r}$
3. Perform an update as directed by (2.5)
4. Repeat steps (2) to (3) until the $\boldsymbol{\theta}$ the algorithms meets one of the defined stopping rules

The update rule was given as

$$\boldsymbol{J}^\intercal \boldsymbol{J} \cdot \delta\boldsymbol{\theta} = \boldsymbol{J}^\intercal \cdot \boldsymbol{r}. \tag{3.1}$$

In this case, the starting value problem refers to the initial choice of the parameter vector $\boldsymbol{\theta}$.

Because the starting values can have a major impact on the effectiveness of the algorithm, a good choice of starting values is vital to achieve satisfactory results.

3.2 Traditional Methods

There have been several approaches proposed over the years to tackle the starting value problem. These range from general best-practices or useful principles, over model specific strategies, to heuristic approaches that derive promising starting values from the raw data. Many of these approaches are also often already implemented in statistical software, such as *STATA*, the *MINPACK* library or packages for the statistical programming language *R*.

One of the most cited source regarding this topic are the five principles proposed by Bates et al.[20], which are based in part on work by Ratkowsky[21]. The first two of these principles recommend to interpret the model function and its derivatives analytically or graphically. This is the most basic form of data analysis and refers to what is casually described as *looking at the data*. In the last three principles the authors describe methods to transform the model function, reduce the dimensions of the problem (also known as *peeling*) or inspect it for the possibility of conditional linearity. These techniques all aim to simplify the model in some way, with the goal to make the estimation of starting values easier.

These principles turned out to be very useful in practice and are generally applicable to nonlinear regression problems. However, because they are only guidelines, they do not guarantee success. Because they rely heavily on human interpretation, the effectiveness of these approaches are also highly dependent on the experience of the person applying them. While an expert statisticians might be able to just *see* good starting values in the plotted data, a novice might not. This makes it also difficult to implement automated versions of these methods. Additionally, the difficulty of applying these approaches scales with the complexity of the problem. Models with more than ten parameters are not uncommon and might make these approaches unfeasible in practice, while data with more than one variable might make visual analysis a lot more difficult.

Aside from those principles, there also haven been numerous strategies proposed in the past to determine starting values for specific model functions. For example, Seber et al. recommend to start with zero for all parameters in logistical models[7]. Schnute et al. described an approach for a special kind of exponential models developed when fitting the growth models of fish[22]. These kind of approaches make starting values less of an issue, but are restricted to a subset of model functions and cannot be extended to other problems. However, this specialization for specific

model types makes them also more suitable to be implemented in an automated way in statistical software.

3.3 Hybrid Approach

In addition to traditional methods, there also exist some nontraditional approaches to the starting value problem, which often (partially) use heuristic methods instead of strictly mathematical ones. One of these methods is the *Hybrid Approach*, which uses genetic algorithms combined with numerical NRA methods to solve the starting value problem.

The Hybrid Approach is a genetic numerical hybrid method. It was proposed by this author in 2014 in his own bachelor's thesis *Numerical Evolution – Using Genetic Hybrid-Methods to Solve Nonlinear Regression Problems*, as an attempt to tackle the starting value problem in a non-traditional way.

3.3.1 Concept

The general idea of using soft-computing methods in regression analysis had its origins in work on the project *Validation of Air Traffic Scenarios On The Example Of The Hamburg Airport*, as part of the *Cluster Of Excellence – Project Efficient Airport 2030* by the city of Hamburg, Germany. The project was a cooperation of several institutions, including the *Hamburg University of Technology*, the Hamburg division of the *German Aerospace Center*, as well as the *University of Hamburg*. One of the goals of the project was to develop a formal model that could be used to estimate the number of passengers the airport has to expect to handle each year.

Using conventional methods – such as regression analysis – turned out to be challenging, due to the high dimensionality of the information, potential nonlinear dependencies, as well as the general problem of proposing an arbitrary model without any basis aside from the raw data. A missing theoretical basis and potential nonlinear parameters generally make it difficult to propose a model and a high amount of variables increases the difficulty of visual inspection and therefore finding reasonable starting values. To allow an automated approach that could evaluate several models in a row without having to manually determine the starting values, it was proposed to investigate the effectiveness of applying genetic algorithms to solve the regression problems without relying on any traditional methods at all.

Although a variant of that approach turned out to be able to solve nonlinear regressions of a particular form of up to 10 parameters, they had an upper limit above which they yielded unsatisfactory results. It was also necessary to use relatively big population sizes, which lead to much slower execution times in comparison to conventional numerical methods.

As a consequence of the aforementioned results, the author concluded that a desired approach would have the following properties:

1. Efficient enough for automated testing
2. Independent of starting values
3. Independent of contextual knowledge

Because the third point would necessitate automated model construction – which is a very complex field of research in itself – the approach was restricted to conform to 1. and 2.

The approach can be summarized by the following train of thought: When viewing the starting value problem through a lens of abstraction, it can conceptually also be seen as a search for a viable combination of starting parameters within a certain parameter space. This means, it can also be interpreted as an optimization problem of finding starting points from which a convergence to a satisfactory extreme point is possible. Because genetic algorithms are agnostic about the problem they are optimizing, they can also be used to optimize the starting values of a given nonlinear problem.

The approach can be summarized by the following steps:

1. Encode the parameters of a nonlinear regression problem into the chromosomes of a genetic algorithm
2. Create a random initial population of starting values
3. Evaluate each chromosome by using its encoded parameters as starting values for a classical numerical method, such as the Gauss-Newton or Levenberg-Marquardt algorithm
4. Use its optimized parameters to calculate the sum of squares and use it to set the fitness of each individual chromosome
5. Use genetic operators to create a new population, which should on average contain better starting values than the previous one
6. Repeat steps 3 to 5 until satisfactory convergence is achieved

3.3.2 Restrictions

Although it was shown in the evaluation that the approach is generally viable, it has several drawbacks. Most prominently, it is restricted to a class of so called *Misc Regression* models, which were defined as

$$f(\boldsymbol{x}, \boldsymbol{\theta}) = y = f_1 + ... + f_n,$$

where each term f_i refers to one of the following parts

$$\begin{aligned}
f_i(x_i, \theta_j) &= \theta_j \cdot x_i && \text{(linear influence)} \\
f_i(x_i, \theta_j) &= \theta_j \cdot \log x_i && \text{(logarithmic influence)} \\
f_i(x_i, \theta_j, \theta_{j+1}) &= \theta_j \cdot x_i^{\theta_{j+1}} && \text{(exponential influence)} \\
f_i(x_i, \theta_j, \theta_{j+1}) &= \theta_j \cdot \theta_{j+1}^{x_i} && \text{(power influence).}
\end{aligned}$$

This restriction was primarily caused by technical limitations. Due to their structure, Misc Regression models are a lot easier to implement than arbitrary model functions and also allow a simple computation of the partial derivatives, without having to use a computer algebra system. Additionally, this class of problems was also suggested by the data of the airport project.

However, as was acknowledged in the bachelor's thesis itself, the class of Misc Regressions is highly restrictive, since only a small number of regression problems actually have that kind of form. In fact, because only very few test problems could be modeled at all, the Hybrid Approach was only tested on generated and highly artificial problems.

The third major drawback is the performance. Because it is a CPU-only solution, larger regression problems can take up to an hour to complete, which makes it unfeasible for many practical applications.

Chapter 4

The Parameter Evolution Approach

The previous chapter provided an overview of several traditional strategies and techniques to tackle the starting value problem. It also showed how the Hybrid Approach attempted to solve it from a nontraditional angle.

By using the presented methods as a basis, this chapter proposes a novel, nontraditional approach to the starting value problem. This is done by first summarizing the advantages and disadvantages of the approaches from the previous chapter and by deriving a set of requirements an ideal solution should comply to. These requirements serve as a basis to define the goals of this thesis, which are then used to construct the approach.

4.1 Overview

Before the approach can be proposed, it is necessary to discuss the rationale behind it and to define the terms under which it will be constructed. This is done by first assessing the existing approaches to the starting value problem – which were presented in the last chapter – and discussing their advantages and disadvantages. On this basis, a set of requirements is derived, which the approach proposed in this thesis shall comply to. Secondly, the goals of the thesis are formally defined. These goals specify what this work is trying to achieve and will be verified in the evaluation. Finally, the general concept of the approach is summarized, before it is discussed in more detail in the following sections of this chapter.

4.1.1 Requirements

To properly convey what the approach proposed in the thesis is trying to achieve, it is necessary to first assess the properties of the existing approaches to the same problem. The previous chapter provided an overview over related work together with their advantages and disadvantages.

Although it was shown that conventional methods can achieve good results under certain circumstances, they also all have some drawbacks that can be summarized as follows:

- No method can be universally applied to all problems
- None of them are fully automated and still require some input from the user
- Some of them are highly dependent on the skills of the user

These are all problems the Hybrid Approach was aiming to solve. Although it was shown that the Hybrid Approach is a promising alternative to conventional methods, it also has some shortcomings:

- It is restricted to a small subset of nonlinear models
- It was only tested on generated problems
- It is too slow for more complex problem

Merging the shortcomings of both traditional methods and the Hybrid Approach leads to the following desired properties of an ideal method:

1. It should be applicable to all problems
2. It should be suitable for automated testing and not require problem specific configurations
3. It should be fast enough for manual and automated testing

Note that these requirements are ideals which an optimal solution would fulfill in every situation. A practical solution probably will not reach complete compliance with these specifications. Additionally, requirements 2. 3. are also dependent on the context. Assessments of terms like "fast enough" or "suitable" might vary depending on the user and the situation.

4.1.2 Goals

Because the Hybrid Approach works in principle and showed some promising results in its evaluation, it is desirable to further investigate the genera idea. The approach in this thesis is based on a similar concept, but aims to generalize its applicability and mitigate its drawbacks. The formal goals of this thesis are defined as follows:

I. Propose an approach to solve nonlinear regression problems independent of starting values with the following properties:
   (i) It is applicable to a wide variety of practical problems
   (ii) It is suitable for automated use and does not require any additional configuration
   (iii) It is suitable for massive parallelism with GPGPU computing and fast enough for practical use

II. Propose and examine different strategies to improve the success rate of the approach

III. Propose a possible implementation for the approach

IV. Design and perform an evaluation that can verify goals I. and II.

4.1.3 Concept

To provide an alternative to the repertoire of existing methods and strategies to tackle the starting value problem, this thesis proposes a generalization and extension of the Hybrid Approach. The goal is to built upon the promising basic idea and to transform it from a proof of concept into a fully developed approach and thereby mitigating its shortcomings and providing solutions for the problems documented in the original bachelor's thesis.

Because the Hybrid Approach was proposed as a proof of concept, it was only very loosely defined. Recalling 3.3.1, the basic steps of the HA were described as the following:

1. Encode the parameters of a nonlinear regression problem into the chromosomes of a genetic algorithm
2. Create a random initial population of starting values
3. Evaluate each chromosome by using its encoded parameters as starting values for a classical numerical method, such as the Gauss-Newton or Levenberg-Marquardt algorithm
4. Use its optimized parameters to calculate the residual and use it to set the fitness of each individual chromosome

5. Use genetic operators to create a new population, which should on average contain better starting values than the previous one

6. Repeat steps 3 to 5 until satisfactory convergence is achieved

Although this description covers all essential steps, it is a strong simplification. In reality, each step has its own intricacies and challenges. Even though such a brief description was sufficient for a proof of concept, it not suitable for an in depth discussion of its individual parts.

To make the approach of this thesis easier to assess, it is broken down into the following three layers:

1. Evolution Layer

2. Evaluation Layer

3. Computation Layer

The *Evolution Layer* contains all components related to genetic algorithms. This is where the chromosomes of the current generation are selected according to their fitness value, to create a new population by applying the genetic crossover and mutation operators.

To determine the fitness of each chromosome, they have to be decoded and evaluated. The evaluation specifies the point at which the approach leaves the domain of the genetic algorithms and the chromosome content is being contextualized for the individual problem. All this is done in the *Evaluation Layer*.

Finally, the evaluation of each chromosome usually encompasses more or less complex calculations. While Evaluation Layer describes the overarching mechanisms that are necessary to evaluate the chromosome data, the actual computations are done in the *Computation Layer*.

The reader may have noticed that the last three paragraphs do not mention the starting value problem or (nonlinear) regression at all. This is because the *Evolution Evaluation Computation (EEC)* layer structure is not bound to regression analysis and could potentially also be used for other types of problems. This idea is explored briefly in the Outlook section of this thesis.

Because of this, using the approach to solve the starting value problem of nonlinear regression problems, can be seen as an instantiation of the more broad EEC concept.

As an explanatory metaphor, the proposed method can then be thought of as a set of starting parameters wandering from the top layer to the bottom layers

and back, to then evolve into new parameters with which the cycle is started anew. This is repeated until satisfactory convergence or a specified stopping rule is reached.

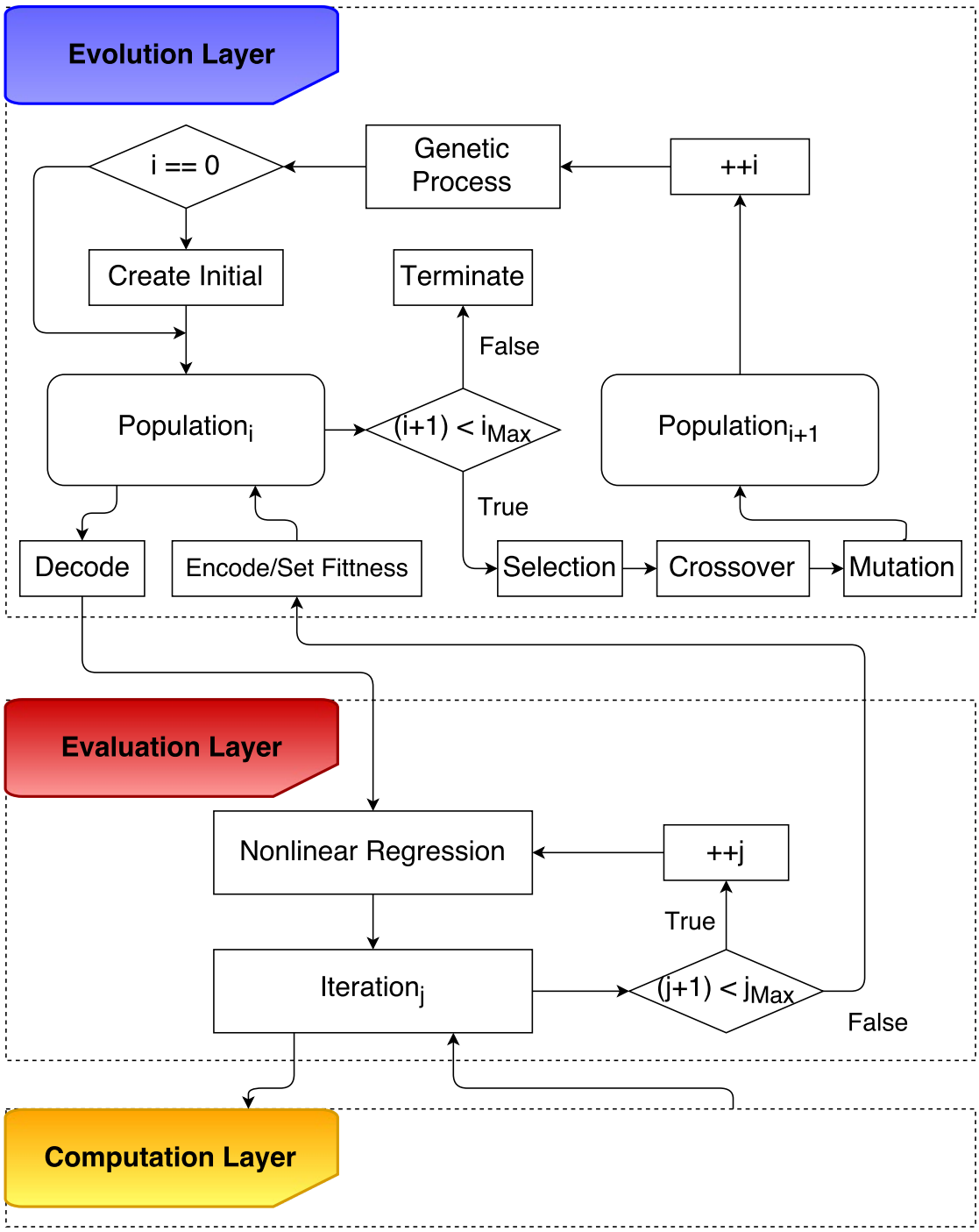

Figure 4.1: The Evolution Evaluation Computation (EEC) layer structure for nonlinear regression problems.

Figure 4.1 visualizes this metaphor. Initially, a set of starting parameters are generated by creating a random population of chromosomes. To set the fitness

value, the encoded parameter data is decoded and then crosses the barrier of the Evolution Layer to the enter the Evaluation Layer, where they are used as starting values for a nonlinear regression algorithm. The regression method itself also transforms the parameters, by sending them to the computation layer where the calculations of each iteration are performed. The iterations are being repeated, until a stopping rule is reached. Afterwards, the result parameters of the NRA method are encoded into the chromosomes, while the summed square difference are set as the fitness value. The chromosomes are then being selected according to the selection strategy and evolve into a new generation of encoded parameters, with which the cycle begins anew until the maximal generation count is reached. This process is applied to every chromosome of the population.

Because of this metaphor, the instantiated EEC approach for the starting value problem of nonlinear regression analysis will be referred to as *Parameter Evolution (PE)*.

What follows is a detailed description of each layer of the PE approach.

4.2 The Evolution Layer

The *Evolution Layer* contains all the parts related to genetic algorithms. In GA terminology, this is denoted by the term genetic process. This layer is also the place where the Parameter Evolution begins, advances and where it is decided when the whole PE algorithm stops.

Figure 4.2 shows how the iterative structure of the genetic process for the PE approach looks like. Each iteration operates on the current population of chromosomes, each containing a set of encoded parameters for the current nonlinear regression problem. In the case of the first iteration, where there is no current population, the chromosomes are created by generating random data.

The next step is to evaluate the current population of chromsomes. But before the encoded parameters contained within the chromosomes can be evaluated, they have to be decoded. Because the genetic process operates on raw binary data, any imaginable encoding could be used to represent the parameters. However, although the type of encoding has no effect on the genetic process, it has a major impact on the effectiveness of the approach. This is covered later in greater detail.

At this point, the encoded parameters cross the layer boundary to be evaluated. An advantage of the EEC structure is that it is not necessary at this point to specify how exactly the evaluation is performed. All that matters is that some kind of evaluation takes place, and that afterwards all chromosomes have their fitness value set, based on the viability of the parameters encoded within them.

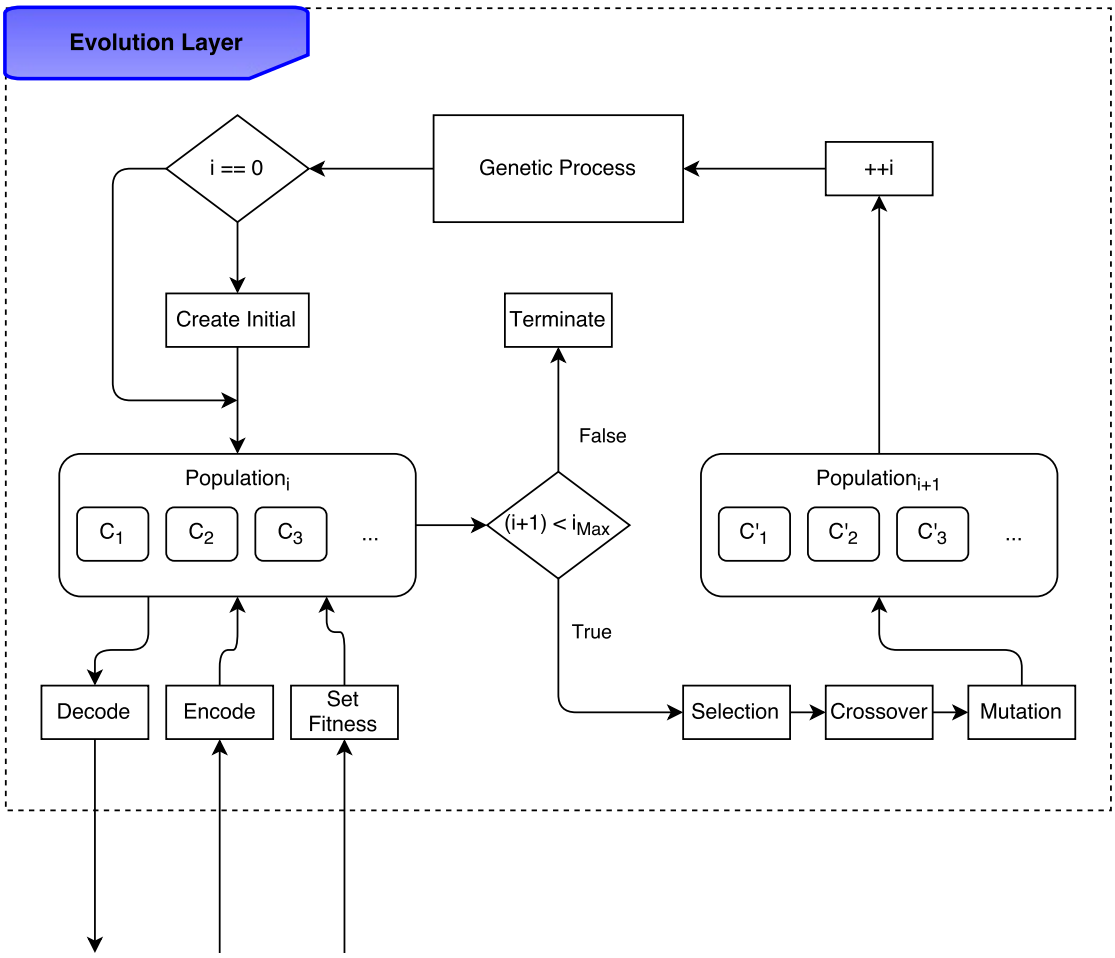

Figure 4.2: Evolution layer of the Parameter Evolution approach.

Here, the PE approach deviates from the traditional design of genetic algorithms. Usually, the evaluation only goes one way. This means that it is a read-only operation, which only determines the fitness value of a chromosome based on its contents, but does not alter the data in any way. In fact, from a software development point of view, a chromosome is often implemented as a *value type*, meaning that its state cannot change after its creation. This is analogous to the original metaphor of genetic algorithms, which is modeled after the real-life genetic processes, in which the DNA does not change within a living organism. Instead, the chromosomes of descendants are always new chromosomes, which are created based on the DNA of the parents.

If one were to strictly follow the original GA metaphor, the genetic process would only set the fitness value of the chromsomes by using the error value returned from the evaluation layer. However, aside from the minimized error value, a nonlinear regression algorithm also returns the approximated result parameters. These parameters themselves can of course also be used again as starting values. It seems reasonable to assume that when the fitness value is high (i.e. the error is low), the

result parameters will even further increase the quality of the encoded chromosome data. This assumption was indeed confirmed for the Hybrid Approach. Therefore, the chromosomes are not only decoded, the result parameters are also encoded after the evaluation.

Provided the maximal generation count is not exceeded, the next step in the genetic process is to create a new population by applying the genetic operators. First, a number chromosomes are selected by using a specific selection strategy. These chromosomes act as the parents for the next generation. There have been a lot of different selection methods proposed over the years and the EEC structure is not limited to any specific one. Some of these strategies are discussed later. The parents are then used as an input to the crossover operation to create children chromosomes, which are in turn applied to the mutation operator. As with the selection, there are many different ways these operations can be implemented.

When the maximal generation count is reached, the algorithm ends and the best parameters can be extracted from the chromosome with the highest fitness.

4.2.1 Chromosome

The *genetic chromosome* is the most basic component of a genetic algorithm. It contains a solution to a particular problem in an encoded representation. The way this solution is encoded may differ from problem to problem. Although not necessarily required, they are classically represented as strings of binary data.

A simple example is the case of a single 32-bit integer number. When using binary chromosomes, one possible choice of storing the data is to use the *trivial encoding*. The trivial encoding just uses the systems internal binary representation of values to encode the data. For 32-bit integers the raw chromosome data could look like this

$$\begin{aligned} C_1 = e_{\text{trivial}}(\ \ 42) &= 00000000000000000000000000101010_2 \\ C_2 = e_{\text{trivial}}(-73) &= 11111111111111111111111110110111_2, \end{aligned}$$

where e_{trivial} denotes the trivial encoding function.

However, in practice the trivial encoding is rarely used, because it usually exhibits bad convergence properties.

A set of chromosomes is called a *population*, while a specific population is also referred to as a *generation*. Chromosomes are the operands to the genetic operators, which are used to create the next generation of chromosomes in the genetic process.

4.2.2 Genetic Operators

There are usually three types of *genetic operators* within the genetic process: The *selection* of chromosomes, the creation of chromosomes by the *crossover* operation and the subsequent *mutation* of the new chromosomes.

Selection

The selection strategy of a genetic algorithm determines which chromosomes are used as the parents for the new generation. The way the chromosomes are selected can differ greatly from method to method, although every strategy uses the fitness value of the chromosomes in one form or another. Exclusively using only the best chromosomes of a population is often not desirable, because GAs tend to prematurely converge when the chromosomes within a population are too similar. The choice of the selection method can have a big influence on the convergence properties of a genetic algorithm. Research in genetic algorithms has produced a great number of strategies over the years. A detailed description and analysis of several methods is presented by Blickle et al[23].

Popular selection methods are:

- **Tournament Selection (TS):** Randomly chooses a number of chromosomes and selects the one with the best fitness. The number of competitors can often be configured
- **Roulette Selection: (RS):** Randomly selects a chromosome while the probability to select a specific individual is proportionally tied to its fitness. A chromosome that has a fitness twice as high as that of another, is also twice as likely to be chosen
- **Linear Rank Selection (LRS):** Randomly selects a chromosome while the probability to select a specific individual is tied to a relative fitness. The LRS uses the absolute fitness values only to sort the population. The probability to choose a specific chromosome then only depends on the relative position of each chromosome to another. An individual in a specific position always has the same probability to be chosen, independently of its absolute fitness value

Because the effect of a strategy can differ from problem to problem, it is often difficult to determine which type of selection is optimal for a particular case. For this reason, finding a suitable method is usually done empirically.

Crossover

The crossover operator creates new chromosomes by combining bit sequences taken from chromosomes of the parent generation. This usually is a binary operation, but there also exist approaches that use three or more operands. The behavior of

the operator can sometimes further be configured to provide additional functionality. For example, some implementations allow the user to specify whether the crossover points should be fixed or dynamic.

Using the crossover operator, the two chromosomes C_1 and C_2 could be used to create a new chromosome

$$\begin{aligned} C_1 = e_{\text{trivial}}(\ \ 42) &= 00000000000000000000000000101010_2 \\ C_2 = e_{\text{trivial}}(-73) &= 11111111111111111111111110110111_2 \\ c(C_1, C_2) = \qquad \widehat{C_1} &= 11111111111111110000000000101010_2, \end{aligned}$$

where c is a fixed crossover operator, which combines the first and last 16 bits of the input chromosomes to create a new one.

Mutation

The final operator is the mutation of a chromosome. Mutations are used to introduce a little bit of random variety to the chromosomes to ensure sufficient diversity within the population. Similar to the crossover method, the mutation sometimes can be configured to accept fixed or non-fixed mutation points. Additionally, often a mutation rate can be set that specifies the probability of a mutation to occur. Although there exist different methods of how the actual mutation is performed, the most popular one is to just flip one or more bits of a chromosome.

The mutation operator is usually applied after the crossover operation to the newly created chromosomes

$$\begin{aligned} \widehat{C_1} &= 11111111111111110000000000101010_2 \\ m(\widehat{C_1}) &= 11111111111111110000100000101010_2, \end{aligned}$$

where m is a mutation operator, which flips a single bit.

4.2.3 Encoding

When using genetic algorithms, one of the most influential parts to achieving satisfying convergence is the choice of how to encode the data into the chromosomes. Because the genetic operators work on the raw chromosome content, the distribution of the data has a major impact on what kind of solutions are constructed for the next generation.

Recalling 4.2.1, a four byte chromosome could look like this:

$$C = 11000000010010010000111111011011_2$$

What these four bytes represent and what encoding is reasonable, is highly dependent on the type of optimization problem the genetic algorithm is used for. In the PE approach, the chromosomes contain a set of parameters in the form of floating point numbers. One of the most straightforward ways to represent floats is to just use the IEEE 754[24] standard, which in many cases is the native representation in software and hardware:

$$\mathrm{d}_{\mathrm{IEEE\_754}}(11000000010010010000111111011011_2) \approx -3.1415927$$

However, this representation is actually not well suited for the use with genetic algorithms, because changing just a single bit can result in either a dramatically different encoded value or almost no change at all, depending on where it occurs:

$$\mathrm{d}_{\mathrm{IEEE\_754}}(11100000010010010000111111011011_2) \approx -5.7952157 \cdot 10^{19}$$
$$\mathrm{d}_{\mathrm{IEEE\_754}}(11000000010010010000111111011010_2) \approx -3.1415925$$

To achieve any kind of convergence, the genetic process is based on the assumption that the likeness of all chromosomes within a population correlates with the similarity of the encoded solutions. The general idea is to start with a heterogeneous population – which means a great diversity of solutions – and then iteratively work towards a more homogeneous population of fitter chromosomes, thereby converging on an (optimal) solution.

A good encoding has the desired property of being robust to changes in the data; meaning that the solution only drastically changes when there is also a drastic change in the chromosome's content.

Discretization

One way of encoding floating point data into chromosomes is to use a discretization method. The *discretized encoding* presented here, maps a floating point value within a certain range to an area of bits

$$\mathrm{e}_{disc}(x, x_{\min}, x_{\max}) = w_{\mathrm{Max}} \cdot \frac{(x - x_{\min})}{(x_{\max} - x_{\min})},$$

$$\mathrm{d}_{disc}(w, x_{\min}, x_{\max}) = \frac{w}{w_{\mathrm{Max}}} \cdot (x_{\max} - x_{\min}) + x_{\min},$$

where x refers to a floating point value within the range $x_{\min}$ and $x_{\max}$ and the maximal representable value of x_{Max}, while w denotes a word of n bytes with the maximal representable value of w_{Max}.

This encoding has several advantages and was successfully used in the implementation of the Hybrid Approach. Not only does it exhibit good distribution properties, it also allows some adaptability through its range selection. Being able to restrict

the encoded data to a minimal and maximal value, guarantees that every new chromosome only contains data within a certain range. This greatly increases the chances of convergence in those cases where it is known in advance that certain solutions are invalid because they are too big or too small. Even in those cases where it is difficult to estimate a restriction, it is still often possible to roughly estimate the orders of magnitude that are sensible. Just excluding very big or small values – e.g everything above 10^{50} or below 10^{-50} – can already lead to much better results. Restricting the range also means that multiple binary representations are mapped to the same value. This increases the robustness of the encoding and results in the chromosomes being more stable for only small changes of data. The encoding can also easily be extended to use a specific binary size. This allows multiple values to be encoded into a single word of n bytes.

Logarithmic

Although the discretized encoding was successfully used with the Hybrid Approach, it has some major flaws in representing certain values. Because the distribution of data is uniform, every value is represented with roughly the same amount of bits. This is problematic, because many regression problems have rather small parameters with values between 0 and 1, and thus need a much finer resolution in that range compared to for example 10 and 100. This issue did not occur during the evaluation of the Hybrid Approach, because the generated test problems did not contain very small parameters.

This could be rectified by just changing the minimal and maximal value for those parameters to the required range. However, if there were an easy way to determine the range of a parameter, the starting value problem would not be an issue. Even roughly estimating the range usually necessitates a manual analysis of the data. This would violate the requirement that was defined in goal I.(ii), which stated that the PE approach should not require any kind of problem-specific configuration.

As an improvement to the discretized encoding, this thesis also proposes a *logarithmic encoding*. This type of encoding does expand upon the discretized encoding by performing an additional logarithmic transformation before discretizing the value:

$$\begin{aligned} \mathrm{e}_{log}(x, x_{\min}, x_{\max}) &= \mathrm{e}_{\mathrm{disc}}(\log_{10}(\mathrm{abs}(x)), 31, x_{\min}, x_{\max}) \vee (\mathrm{sgn}(x) \cdot 2^{31}), \\ \mathrm{d}_{log}(w, x_{\min}, x_{\max}) &= (w \wedge 2^{31}) \vee 10^{\mathrm{d}_{\mathrm{disc}}(w, 31, x_{\min}, x_{\max})} \end{aligned}$$

The general idea is to transform the value to a $x = 10^{\hat{x}}$ representation and store $\hat{x}$ instead of x itself. Although this encoding has the potential weakness of using a single dedicated bit for the sign – which can easily be flipped – it should also provide additional stability by being able to cover the same range of the discretized encoding with an overall much smaller value.

Note that the logarithmic encoding requires a variant of d_{disc} that allows to also specify a binary size.

4.3 The Evaluation Layer

The Evaluation Layer contains all parts relevant to the evaluation of the chromosomes.

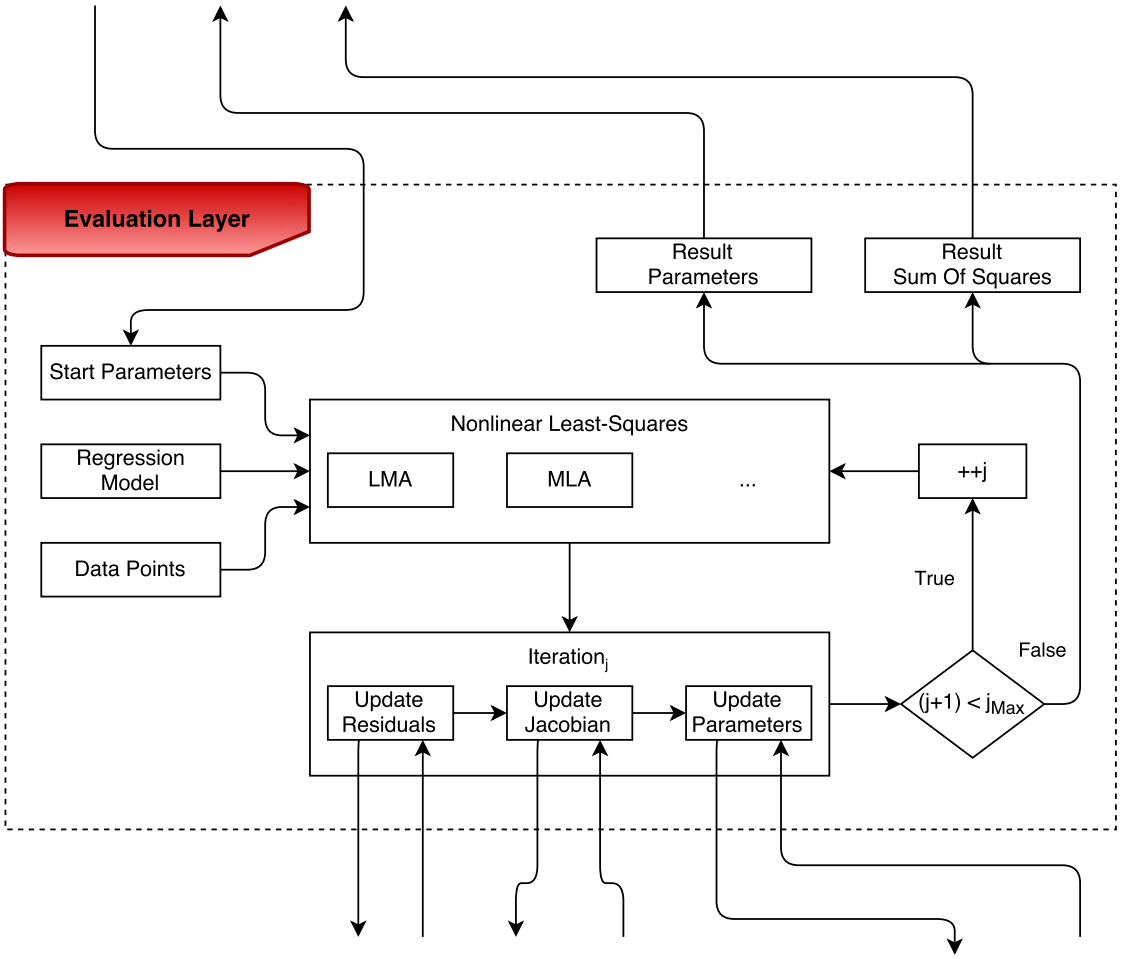

Figure 4.3: Evaluation layer of the Parameter Evolution approach.

Figure 4.3 shows the layer for the PE approach. The evaluation begins by using the parameters as starting values for a traditional nonlinear regression method. The methods covered here are all based on the nonlinear least-squares approach. Although they are different in detail, they all share a similar iterative structure. The evaluation layer hereby only describes the individual steps, while the techniques used to do the actual computation are part of the Computation Layer. As with most parts of the PE approach, there is no requirement for a specific NRA algorithm. However, similar to the chromosome encoding, not all variants are suited

for the PE approach and the choice can have a major impact on the overall success rate of the whole method.

After the NLS method has terminated, the Evaluation Layer returns the lowest residual error and the parameters with which it was attained.

4.3.1 Nonlinear Least Squares

The PE approach is not limited to using a specific nonlinear regression method. Any iterative algorithm suitable for nonlinear regression analysis, that is based on specifying initial parameters, can be used. However, the methods that are presented here are all based on the nonlinear least squares approach as presented in 2.2.2. The main reason for this choice is that they were used in the evaluation of the Hybrid Approach and therefore already have been confirmed to be suited to the approach presented here. They are also among the most popular NRA methods and can be applied to a wide variety of problems, which is required to fulfill goal I.(i).

In particular, this thesis focuses on the Levenberg-Marquardt and the Modified Levenberg algorithms.

Levenberg-Marquardt And Modified Levenberg Algorithm

The *Levenberg-Marquardt Algorithm (LMA)* is based on the Gauss-Newton algorithm presented in 2.2.3. Although the GNA has good convergence properties when the initial guess is near a minimum, the method diverges if it is too far away or if the matrix $\boldsymbol{J}^\intercal\boldsymbol{J}$ is ill-conditioned. One of the several approaches to improve the Gauss-Newton algorithm was made by Kenneth Levenberg in 1944[25]. He suggested to add an *dampening-term* to the update rule (2.5)

$$(\boldsymbol{J}^\intercal\boldsymbol{J} + \lambda\boldsymbol{I}) \cdot \delta\boldsymbol{\theta} = \boldsymbol{J}^\intercal \cdot \boldsymbol{r}. \tag{4.1}$$

The non-negative *dampening factor* $\lambda\boldsymbol{I}$ is adjusted at each iteration. If the reduction of the sum of squares S is insufficient in the residual, λ can be increased giving a step closer to the descent direction of the gradient, whereas if the convergence rate is high, a low value brings the method closer to the Gauss-Newton algorithm. In 1963, Donald Marquardt further improved the method by scaling each component of the gradient according to the curvature, so that there is larger movement along the direction where the gradient is smaller[26]. This avoids slow convergence in the direction of a small gradient. He replaced the identity matrix $\boldsymbol{I}$, with the diagonal matrix consisting of the diagonal elements of $\boldsymbol{J}^\intercal\boldsymbol{J}$

$$(\boldsymbol{J}^\intercal\boldsymbol{J} + \lambda \cdot \operatorname{diag}(\boldsymbol{J}^\intercal\boldsymbol{J})) \cdot \delta\boldsymbol{\theta} = \boldsymbol{J}^\intercal \cdot \boldsymbol{r}. \tag{4.2}$$

At this point, the PE approach again diverges to an unusual path. Traditionally, Marquardt's update rule 4.2 is considered to be a straight up improvement over Levenberg's 4.1, because it increases the amount of movement in directions where convergence is certain. However, if the starting values are bad, using Maquardt's rule can lead to instability and numerical problems. In most conventional uses of the LMA, this a non-issue, because the starting values need to be manually selected and are usually not completely off. For this reason, a statistician usually prefers a much faster converging method to one that is only better in rare cases.

The PE approach on the other hand favors different characteristics. Bad starting values are almost guaranteed to happen in the first few generations and because the results are encoded and used for the next generation of starting values, stability is much more important. Additionally, it is not critical if a set of starting values does not fully converge in a single run of the NRA algorithm, because the process is repeated multiple times.

For this reason, the PE approach also explores the viability of using Levenberg's update rule in addition to the one from Marquardt. Because the method is not exactly the same as the Levenberg algorithm, it will be referred to by the name *Modified Levenberg Algorithm (MLA)*.

Because they only differ in the update rule, both algorithms can be described as follows:

1. Guess appropriate starting parameters for $\boldsymbol{\theta}$
2. Compute the current values for the Jacobian $\boldsymbol{J}$ and the residual vector $\boldsymbol{r}$
3. Do an update as directed by either (4.1) (MLA) or (4.2) (LMA)
4. Evaluate the error of the new parameter vector
5. If the error has increased, then retract the step and increase λ by a constant factor
6. If the error has decreased as a result of the update, then accept the step and decrease λ by a constant factor
7. Repeat steps (2) to (6) until the $\boldsymbol{\theta}$ is at the desired accuracy or the algorithm meets one of the defined stopping rules

Both algorithms share three essential operations that have to be applied on every iteration:

1. Update the residual vector
2. Update the Jacobian matrix
3. Update the parameter vector

Residual Update

Recalling 2.2.2, the residuals were described as a vector $\boldsymbol{r}$ with each entry defined as

$$r_k = y_k - f(\boldsymbol{x}_k, \boldsymbol{\theta}),$$

with y_k and $\boldsymbol{x}_k$ referring to the k'th data point, $\boldsymbol{\theta}$ to the parameter vector and f to the model function.

To perform the residual update, each entry r_k has to be calculated for every data point k with the parameter vector of the h'th iteration of the NLS algorithm

$$\boldsymbol{r} = \boldsymbol{y} - \hat{f}(\boldsymbol{X}, \boldsymbol{\theta}_h),$$

where $\boldsymbol{y}$ and $\boldsymbol{X}$ describe all data points, $\boldsymbol{\theta}_h$ the h'th parameter vector and $\hat{f}$ the vector-valued model function.

Jacobian Update

The Jacobian $\hat{\boldsymbol{J}}$ is the matrix of all first-order partial derivatives of a vector of functions $\boldsymbol{f}$. In the context of regression analysis, this means that the model function f can be instantiated with every data point k, therefore defining $\boldsymbol{f}$ as a K-vector of functions

$$f_k(\boldsymbol{\theta}) \coloneqq f(\boldsymbol{x}_k, \boldsymbol{\theta})$$
$$\boldsymbol{f}(\boldsymbol{\theta}) \coloneqq \begin{bmatrix} f_0 \\ \vdots \\ f_k \end{bmatrix}.$$

This leads to the definition of the Jacobian as

$$\hat{\boldsymbol{J}} = \begin{bmatrix} \frac{\partial \mathbf{f}}{\partial \theta_1} & \cdots & \frac{\partial \mathbf{f}}{\partial \theta_M} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial \theta_1} & \cdots & \frac{\partial f_1}{\partial \theta_M} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_K}{\partial \theta_1} & \cdots & \frac{\partial f_K}{\partial \theta_M} \end{bmatrix}.$$

However, when using the Jacobian in NRA methods, the term actually refers to the matrix of calculated Jacobian entries

$$\boldsymbol{J} = \begin{bmatrix} \hat{\boldsymbol{J}}_{1,1}(\boldsymbol{\theta}_h) & \cdots & \hat{\boldsymbol{J}}_{1,M}(\boldsymbol{\theta}_h) \\ \vdots & \ddots & \vdots \\ \hat{\boldsymbol{J}}_{K,1}(\boldsymbol{\theta}_h) & \cdots & \hat{\boldsymbol{J}}_{K,M}(\boldsymbol{\theta}_h) \end{bmatrix},$$

where $\boldsymbol{\theta}_h$ are the parameters of the h'th iteration of the NRA algorithm.

Parameter Update

One crucial aspect of both the LMA and MLA is how to perform the update of the parameter vector $\boldsymbol{\theta}$. The update rule of the MLA algorithm was given as follows

$$(\boldsymbol{J}^\intercal \boldsymbol{J} + \lambda \cdot \boldsymbol{I}) \cdot \delta\boldsymbol{\theta} = \boldsymbol{J}^\intercal \cdot \boldsymbol{r}. \tag{4.3}$$

To apply the update, the equation has to be solved for $\delta\boldsymbol{\theta}$. Substituting $\boldsymbol{B}$ for $(\boldsymbol{J}^\intercal \boldsymbol{J} + \lambda \cdot \boldsymbol{I})$ yields

$$\begin{aligned}
(\boldsymbol{J}^\intercal \boldsymbol{J} + \lambda \cdot \boldsymbol{I}) \cdot \delta\boldsymbol{\theta} &= \boldsymbol{J}^\intercal \cdot \boldsymbol{r} \\
\boldsymbol{B} \cdot \delta\boldsymbol{\theta} &= \boldsymbol{J}^\intercal \cdot \boldsymbol{r} \\
\boldsymbol{B}^{-1} \cdot \boldsymbol{B} \cdot \delta\boldsymbol{\theta} &= \boldsymbol{B}^{-1} \cdot \boldsymbol{J}^\intercal \cdot \boldsymbol{r} \\
\boldsymbol{I} \cdot \delta\boldsymbol{\theta} &= \boldsymbol{B}^{-1} \cdot \boldsymbol{J}^\intercal \cdot \boldsymbol{r} \\
\delta\boldsymbol{\theta} &= \boldsymbol{B}^{-1} \cdot \boldsymbol{J}^\intercal \cdot \boldsymbol{r}.
\end{aligned}$$

At this point a problem arises, because not every matrix is invertible. Quite to the contrary, most matrices $\boldsymbol{B}$ that occur in the LMA or MLA algorithms are *singular*, meaning that they are not invertible.

One way to solve this problem is to use the Moore-Penrose pseudoinverse $\boldsymbol{B}^+$ of the matrix instead of the actual inverse $\boldsymbol{B}^{-1}$. The parameter update $\delta\boldsymbol{\theta}$ can then be calculated as

$$\begin{aligned}
\delta\boldsymbol{\theta} &= \boldsymbol{B}^+ \cdot \boldsymbol{J}^\intercal \cdot \boldsymbol{r} \\
\delta\boldsymbol{\theta} &= (\boldsymbol{J}^\intercal \boldsymbol{J} + \lambda \cdot \boldsymbol{I})^+ \cdot \boldsymbol{J}^\intercal \cdot \boldsymbol{r}.
\end{aligned}$$

There are several ways the pseudoinverse can be constructed, most prominently by using either a Cholesky or QR decomposition of the matrix. These methods have the advantage that they are comparatively cheap to compute and still stable enough for many practical applications. However, because in the PE approach the first generation of chromosomes usually contains bad starting values, the decomposition is very sensitive to numerical problems and the stability provided by those methods is usually insufficient. This was confirmed in the evaluation of the Hybrid Approach, where it was shown that the stability of the used decomposition method is vital and that only the singular value decomposition was able to solve more difficult problems. For this reason, this thesis will only focus on the SVD, although the PE approach can in principle also be used with a different method.

After the SVD of the matrix $\boldsymbol{B}$ has been obtained, the psuedoinverse can be determined by

$$\boldsymbol{B}^+ = \boldsymbol{V}\boldsymbol{\Sigma}^+\boldsymbol{U}^\intercal,$$

where $\mathbf{\Sigma}^+$ is the pseudoinverse of $\mathbf{\Sigma}$, which can be determined by replacing every non-zero diagonal entry by its reciprocal and afterwards transposing the resulting matrix.

4.4 The Computation Layer

The *Computation Layer* contains those parts of the PE approach that are performing the actual computations on the data. The vast majority of work is done here.

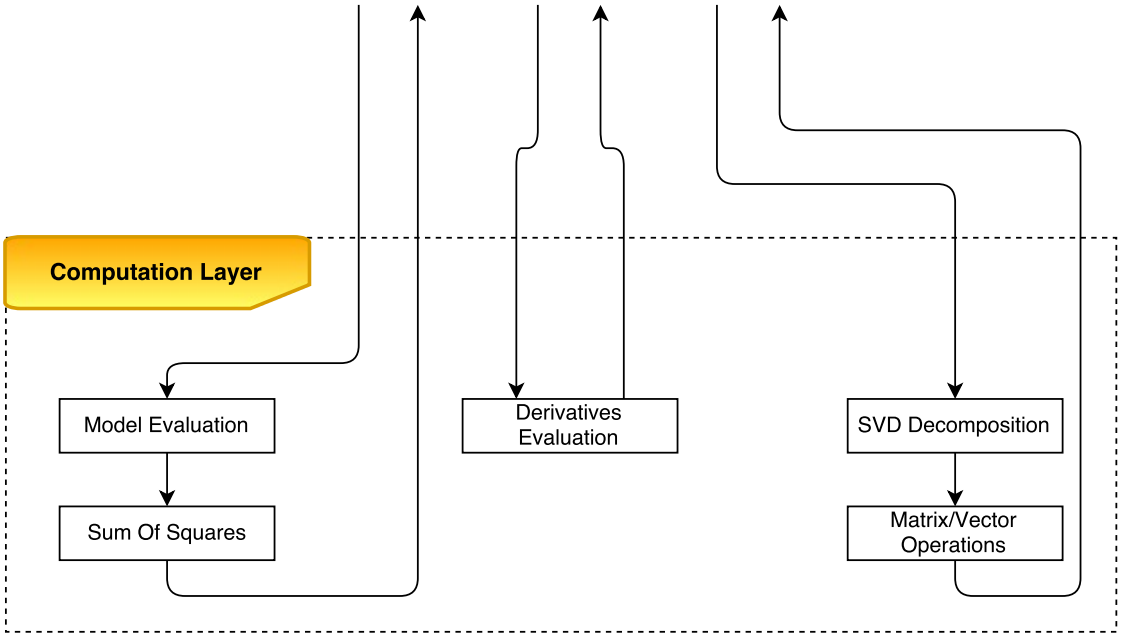

Figure 4.4: Computation layer of the Parameter Evolution approach.

Figure 4.4 shows how the computation layer of the PE approach looks like. Contrary to the other layers, the components of this layer are not all part of the same process. Aside from the SVD algorithm, which is one of the most important parts of the approach, it is rather thin from a theoretical point of view, because most components just perform the actual calculations from each step of the NRA algorithm.

The computation layer is discussed in greater detail in the next chapter, when the approach is implemented.

4.4.1 Singular Value Decomposition

Aside from its performance, other important properties of a numerical algorithm are its *stability*, *accuracy* and *robustness*. The exact definitions of the last three terms vary and often depend on the context. In numerical linear algebra, the stability generally refers to the difference between the *true* solution of a problem and the solution approximated by the algorithm[27].

An algorithm is considered stable, when it is less affected or can even damp out small errors in the input data, while an instable algorithm might magnify the errors. Additionally, an algorithm is considered robust when it does create similar results for very small changes in the input data. Finally, the accuracy of an algorithm refers to the chances of introducing errors caused by numerical problems, such as rounding or insufficient floating point precision.

In this thesis, these three terms are referred to as the *numerical properties* of an algorithm. The numerical properties of an algorithm are considered good, when it has a high accuracy and is stable and robust.

The numerical properties of the methods used in the PE approach are vital to achieve good results. Because the initial starting values will most likely be poor, it is important that the NRA method does converge to something, even if it is a bad solution. If the numerical properties are bad, the NRA method might produce invalid floating point values – also known as *not a number (NaN)* values – which must be prevented at all costs, because they cannot be used as parents for a new generation of parameters.

As mentioned in chapter 2.1.3, there are several ways of how the SVD can be calculated. The *Handbook of Linear Algebrar (HLA)* alone lists seven algorithms to obtain the SVD of a matrix[28], which differ greatly in their efficiency and numerical properties.

One of the most widely used methods is the calculation by *Householder reduction to bidiagonal form*. It is implemented in many popular tools, such as MATLAB oder MAPLE, due to its efficiency compared to other methods, while still having sufficient numerical properties for most problems. However, due to the requirements for maximal numerical stability, it is not ideal for the PE approach.

Instead, this approach will use the *Jacobi Rotation SVD*, which is considered to be among the best algorithms in regards to numerical properties, while at the same time also being one of the most computationally expensive variations.

The algorithm is defined in the HLA as follows:

Algorithm 7: Jacobi Rotation SVD:
Input: m, n, A where A is $m \times n$.
Output: Σ, U, V so that Σ is diagonal, U and V have orthonormal columns, U is $m \times n$, V is $n \times n$, and $A = U\Sigma V^T$.

1. $B \leftarrow A$. (This step can be omitted if A is to be overwritten with B.)
2. $U = I_{m \times n}$.
3. $V = I_{n \times n}$.
4. If $m > n$, compute the QR factorization of B using Householder matrices so that $B \leftarrow QA$, where B is upper triangular, and let $U \leftarrow UQ$. (See Algorithm 6 for details.)
5. Set $N^2 = \sum_{i=1}^{n} \sum_{j=1}^{n} b_{i,j}^2$, $s = 0$, and *first* = **true**.
6. Repeat until $s \leq \varepsilon^2 N^2$ and *first* = **false**.
   a. Set $s = 0$ and *first* = **false**.
   b. For $i = 1, \ldots, n-1$
      i. For $j = i+1, \ldots, n$:
      - $s = s + b_{i,j}^2 + b_{j,i}^2$.
      - Determine $d_1, d_2, c = \cos(\theta)$ and $s = \sin(\varphi)$ with the property that d_1 and d_2 are positive and

$$\begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} b_{i,i} & b_{i,j} \\ b_{j,i} & b_{j,j} \end{bmatrix} \begin{bmatrix} \hat{c} & \hat{s} \\ -\hat{s} & \hat{c} \end{bmatrix} = \begin{bmatrix} d_1 & 0 \\ 0 & d_2 \end{bmatrix}.$$

      - $B \leftarrow R_{i,j}(c, s) B R_{i,j}(\hat{c}, -\hat{s})$ where $R_{i,j}(c, s)$ is the Givens rotation matrix that acts on rows i and j during left multiplication and $R_{i,j}(\hat{c}, -\hat{s})$ is the Givens rotation matrix that acts on columns i and j during right multiplication.
      - $U \leftarrow U R_{i,j}(c, s)$.
      - $V \leftarrow V R_{i,j}(\hat{c}, \hat{s})$.
7. Set Σ to the diagonal portion of B.

Figure 4.5: The Jacobi Rotation SVD algorithm as it is defined in the Handbook of Linear Algebra.

Although many parts of the algorithm are straightforward, one crucial detail is not specified. In step 6 b., the work matrix $\boldsymbol{B}$ has to be updated by applying a left and a right rotation to itself. The corresponding angles θ and ϕ have to be determined by solving the linear system

$$\begin{bmatrix} d_1 & 0 \\ 0 & d_2 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \cdot \begin{bmatrix} b_{i,i} & b_{i,j} \\ b_{j,i} & b_{j,j} \end{bmatrix} \cdot \begin{bmatrix} \cos\phi & \sin\phi \\ -\sin\phi & \cos\phi \end{bmatrix}, \tag{4.4}$$

where d_1 and d_2 have to be positive values.

There are several ways to solve this. One option is based on the theorem that any 2×2 matrix can be decomposed into a rotation, a nonuniform scale, and another rotation[29]:

$$\begin{bmatrix} b_{i,i} & b_{i,j} \\ b_{j,i} & b_{j,j} \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \cdot \begin{bmatrix} d_1 & 0 \\ 0 & d_2 \end{bmatrix} \cdot \begin{bmatrix} \cos\phi & \sin\phi \\ -\sin\phi & \cos\phi \end{bmatrix}^{\mathsf{T}}$$

Because the two rotation matrices are unitary – i.e. their transpose is also their inverse – the equation can be rearranged to 4.4.

The rotation angles θ and ϕ, as well as the diagonal entries d_1 and d_2, can be determined as follows:

$$E = \frac{b_{i,i} + b_{j,j}}{2}, \quad F = \frac{b_{i,i} - b_{j,j}}{2}, \quad G = \frac{b_{j,i} + b_{i,j}}{2}, \quad H = \frac{b_{j,i} + b_{i,j}}{2}$$

$$Q = \sqrt{E^2 + H^2}, \quad R = \sqrt{F^2 + G^2}$$

$$d_1 = Q + R, \quad d_2 = Q - R$$

$$a_1 = \operatorname{atan2}(G, F), \quad a_2 = \operatorname{atan2}(H, F),$$

$$\theta = \frac{a_2 - a_1}{2}, \quad \phi = \frac{a_2 + a_1}{2}$$

Chapter 5

Implementation

The previous chapter introduced the PE approach as a nontraditional method to solve the starting value problem. The approach was presented as an instantiation of the EEC layer structure.

After the PE approach was described from a theoretical point of view, this chapter presents an exemplary implementation of the approach. This is done by going through each layer and presenting those components that are critical to the approach, thereby discussion the individual challenges of implementing it in such a way that it complies with the goals set in the previous chapter.

5.1 Overview

The EEC is not only beneficial to structure and visualize the PE approach, it is also helpful in developing the architectural design of its implementation.

What follows is a description of all important components within each layer of the PE approach. However, due to the size and complexity of the implementation, it is not feasible to describe every component in detail. Instead, many components will be presented in a more general and at times simplified way. These descriptions in some cases differ in details from the actual implementation used in the evaluation. Please note that although the components share some similarities with those described for the Hybrid Approach, they are completely redesigned and not part of the Soft Computing Framework library that was used to implement the HA.

As with the Hybrid Approach, the PE approach is implemented in C++; a language that is well suited for the approach, due to its focus on high performance in combination with higher-level abstractions.

To make the source code listings more readable, they are usually presented as abridged versions of the original implementations. Constructors and assignment operators in class declarations are usually omitted, unless they are of particular

interest. Additionally, functions are shown without `const` qualifiers in the parameters. Additional omissions are denoted by ellipses in the source code.

5.1.1 Custom Implementation vs. Libraries

Before designing anything, it is paramount to first assess what software components are required and to determine whether the needed functionality is already implemented in any existing library. Using libraries over a custom implementation can have several advantages. Unless a library is brand-new or is very niche and has only few users, it is usually less error-prone, because it stems from a more mature codebase that has already been tested by several people. Additionally, a library might have been developed by experts in the corresponding fields and it is very unlikely that a custom solution outperforms something that has been in development for several years – or in some cases decades. This is especially true for hybrid solutions like the PE approach that consist of several components from different fields of research, which are already quite complex on their own.

Due to the division in several layers, it is pretty straight-forward to determine the necessary components of the PE approach; which in many cases can basically be taken directly from the figures of each individual layer from the previous chapter.

On the top-level, the required functionality can be summarized as follows:

- A genetic algorithm, including its required sub components
- The nonlinear regression methods LMA and MLA
- A representation of the regression model
- Vectors and matrices, including their operations
- The Jacobi SVD method

Even though there are some excellent libraries for each of these points, no combination fits the requirements of the PE approach. This is mainly due to the unusual nature of the approach, which requires very fine control over specific parts of the implementation. There are also certain aspects of the approach that go against the paradigms some of these methods are based on. For example, most GA libraries provide no way to encode data into a chromosome and instead implement them as value types. The NRA libraries have a similar problem in that they are usually not designed for maximal numerical stability, which is an important attribute for the PE approach. Other parts, such as the MLA method, are not available at all in popular libraries.

However, the biggest problem poses the desired support for massive parallelism using GPU acceleration. What all of the current GPGPU solutions have in common, is that they are using a custom language to program the kernel of the GPU

program. C++ AMP offers some advantages in this respect, because it is based on a subset of C++. But even with AMP, it is not possible to simply reuse existing libraries on the GPU. Although there exist some libraries for OpenCL and especially CUDA that offer some of the needed functionality, they are also not suited for the PE approach, because they are designed for a different area of application. Usually, the motivation to use GPGPU computing over CPU computing is that a single problem gets too large to be solvable efficiently on a CPU. The GPGPU algorithm then splits the problem into independent chunks that can be computed in parallel. The PE approach on the other hand aims to utilize the GPU to compute a great number of comparatively small problems at once. For example, this is why the PE approach does not benefit from using GPU implementations for the computation of the SVD, because those are usually designed to be applied to a single very large matrix with several thousand rows and columns, instead of many relatively small matrices in parallel. Additionally, many of the GPGPU implementations are themselves experimental and do not provide the quality or support needed for being incorporated into a bigger project.

After some thorough consideration, these reasons lead to the decision of using a custom implementation rather than third-party libraries.

5.1.2 Genericty vs. Polymorphism

As was shown in chapter 4, the PE approach has an abstract structure and contains many parts that can be filled with different implementations for the particular methods and techniques that are used by the general algorithm.

From a software development point of view, there are several ways this can be reflected within the implementation. An obvious option to abstract from a particular implementation is to use *polymorphism*. This would allow the core components to work on interface classes and to provide the user with the option to specify an arbitrary implementation with the desired functionality. However, because the implementation of the PE approach is done in C++, another option is to use *genericity* or *generic programming*. Through the usage of *templates*, C++ allows classes and functions to be parameterized with additional types and values. It is important to note that although the two techniques share a subset of common use, they are not intended to solve the exact problems and generally have very different areas of applications.

The main difference between the two techniques is that polymorphism gets resolved at run time, while templates are resolved at compile time. This results in both advantages and disadvantages, depending on the area of application. Polymorphism uses *virtual functions*, which are only known at the point of invocation. This allows functionality to be changed while the program is running. But the very reason that it not known what particular function is called not only introduces overhead for the lookup, it also prevents many types of code optimizations

by the compiler (e.g.*inlining*). Templates on the other hand are resolved at compile time and therefore cannot be changed after the program was built. However, because the types are known, the compiler can resolve the function calls and can use all types of optimization at its disposal. This difference can have a significant impact on performance critical applications.

A necessary condition to choose genericity over polymorphism is that there is no need to interchange or mix components during run time; which is the case for the PE approach. Another important factor to consider is that C++ AMP is based on a subset of the C++ language that does not allow virtual functions to be used. This means that every component intended to be used with C++ AMP would require to be implemented without polymorphism anyway.

For this reason and because the implementation of the PE approach is performance critical, the implementation will rely on genericity instead of polymorphism.

5.1.3 GPGPU vs. CPU

The reader may have noticed that although exploring the feasibility of GPGPU computing is one of the goals of this thesis, there has been little mention of it up to this point. The reason for this is that the PE approach itself is not specific to any kind of hardware. Everything described in the previous chapter could also be implemented and parallelized on a CPU.

However, because the genetic process is comparatively cheap, the vast majority of the work is done in the evaluation of the current population, which is roughly the same type of work for every single chromosome. And since populations can consist of hundreds or even thousands of chromosomes, the PE approach has the potential to greatly benefit from massive parallelism using a GPU.

Additionally, it is important to consider the intended area of application of the PE approach. Although the argument from the last paragraph means that the approach can also be parallelized very well on a big CPU cluster – with potentially even more cores than a GPU – it is rarely the case that a statistician has access to such hardware; mainly because conventional nonlinear regression methods methods do not require nearly as much computational prowess. Consumer grade GPUs on the other hand are comparatively cheap, with costs of generally less than 1000 US Dollars, even for the high-end models. Even if the consumer variants are not sufficient – for example because a higher double-precision performance is needed – professional GPUs are still a lot more affordable when compared to a computing center.

5.2 C++ AMP

Section 2.4 provided an overview of some available technologies to utilize GPGPU computing, including their advantages and disadvantages. The main reason why the implementation of the PE approach is using C++ AMP over the other methods, is its compatibility with C++. Because the PE approach is heterogeneous, there is a constant transfer of data between the CPU and GPU. Being able to use a single language greatly reduces the complexity of integrating the GPU interaction into the other parts of the software. Additionally, being able to use higher level concepts, such as genericity and lambdas, makes it easier to design the GPGPU components.

It is also important to note that although C++ AMP is mostly used for GPGPU computing, it is agnostic about the specific hardware it runs on. Instead, AMP uses the concept of an *accelerator*. An accelerator can be a GPU, but it could also refer to different types of hardware, such as a computation cluster or a FPGA. In Microsoft's implementation of AMP, an accelerator can be any DirectX 11 compatible physical device or even a suitable *virtual accelerator*, such as Microsoft's *Windows Advanced Rasterization Platform (WARP)*. WARP emulates DirectX 11 on a CPU, while trying to utilize *vector instructions* as much as possible. This makes AMP very flexible in benefiting from different types of hardware. Unless otherwise specified, AMP will always use the *default accelerator*, which in most cases is the most power physical device available. If no physical accelerator can be used, either because none is installed or it is not capable of DirectX 11, AMP will automatically use WARP as a fallback option. This guarantees that an AMP program can be executed, even if no physical accelerator is present.

5.2.1 Basics

One advantage of AMP is that it is comparatively easy to use and to integrate into a C++ program. Provided the compiler has an AMP implementation, a simple example can be expressed by only a few lines of code.

Listing 5.1: C++ AMP example: Function that adds the values of two arrays

```
1 #include <amp.h>
2 using namespace concurrency;
3
4 void add_arrays(int n, int* pA, int* pB, int* const pC)
5 {
6   array_view<int, 1> a(n, pA);
7   array_view<int, 1> b(n, pB);
8   array_view<int, 1> c(n, pC);
9
10  parallel_for_each(c.extent, [=](index<1> idx) restrict(amp)
11  {
12    c[idx] = a[idx] + b[idx];
13  });
14 }
```

Listing 5.1 shows a simple example of a function that adds the values of two arrays.

The accelerator code – the so called *kernel* – is specified as an argument to the `parallel_for_each` function in the `concurrency` namespace. The kernel can be specified as a pointer to a standalone function, a reference to a functor or as a lambda function, as shown in this example. Note that this is the complete code that is necessary for `add_arrays` to be called from within a C++ program and for the kernel to be executed on an accelerator. Unless otherwise specified, AMP will automatically execute each kernel argument of a `parallel_for_each` call on the default accelerator.

Another important aspect to note is that the example does not contain any explicit copy operations, which specify when which data is transferred to the accelerator and back. Instead, the data is transfered implicitly, by using the `array_view` type. An `array_view` is a wrapper class for raw data pointers or containers from the C++ standard library. Because the variables `a`, `b` and `c` are captured by the lambda (as specified by the `[=]` syntax), the `array_view` automatically transfers the data to the accelerator. When the variables go out of scope – which in C++ calls the destructor of the class – AMP automatically detects that the content of `c` has changed and synchronizes the data to the memory location pointed to by `pC`. This allows effortless data transfers without having to manually handle synchronization. However, it should be noted that using `array_view` is optional and that AMP also offers manual control over copy and synchronization operations.

Finally, the example shows one of the only two keywords C++ AMP adds to the language. The expression `restrict(amp)` specifies that the definition of the lambda is intended for AMP use, meaning that it can be executed on an accelerator. Every function that is called from the kernel, either directly or indirectly, has to be qualified this way. In doing so, the user communicates to the compiler that the function conforms to the restrictions AMP imposes on accelerator code[30]. Violating any of those restrictions will result in a compiler error. Most of the restrictions are based on actual physical differences many accelerators have compared to CPUs. For example, a GPU usually does not have a call stack in the classical sense. It is therefore not possible to perform recursion the same way as on the CPU. Additionally, although GPUs have global memory, they usually do not support the concept of a heap, which means that everything that relies on dynamic memory allocation cannot be used. This includes language features such as virtual functions, function pointers, polymorphism and others. The restrict keyword can also be specified as `restrict(cpu)` and `restrict(cpu, amp)`. Because it is part of the signature of a function, the keyword can be used to overload functions and use the corresponding version depending on whether it is called from within or from without an AMP kernel.

5.2.2 Tiling

By default, C++ AMP will automatically determine how many threads are executed in parallel. This is often not ideal, because the amount of threads that

can effectively run concurrently is strongly dependent on the actual hardware and AMP only works on the abstract concept of an accelerator. To provide additionally flexibility to the developer, AMP offers a functionality known as *tiling*. Tiling allows the user to explicitly specify how many threads are executed at once.

Listing 5.2: C++ AMP example: Tiled function that adds the values of two arrays

```
1 #include <amp.h>
2 using namespace concurrency;
3 constexpr int TileSize = 32;
4
5 void add_arrays_tiled(int n, int* pA, int* pB, int* pC)
6 {
7   array_view<int, 1> a(n, pA);
8   array_view<int, 1> b(n, pB);
9   array_view<int, 1> c(n, pC);
10
11  parallel_for_each(c.extent.tile<TileSize>(),
12  [=](tiled_index<TileSize> tidx) restrict(amp)
13  {
14    c[tidx.global[0]] = a[tidx.global[0]] + b[tidx.global[0]];
15  });
16 }
```

Listing 5.2 shows how the kernel of the first example can be modified to use tiling. Both functions only differ in the usage of the `tiled_index` instead of the regular `index`. Tiling is usually used to divide a larger problem into smaller parts that are then processed in parallel. This is different from the PE approach, where all chromosomes are evaluated totally independent from one another.

However, tiling also allows the usage of `tile_static` memory. Like CPUs, many accelerators, most prominently GPUs, have several levels of caches to speedup memory access. But contrary to a CPU, usually not all caches are utilized automatically. Instead, the caches are programmable, allowing a much more finer control of how the caches are used. To use the programmable cache in AMP, a variable can be qualified by using the keyword `tile_static`. A tile static variable is shared among all threads of the current tile.

Taking full advantage of the tile static memory is a complex topic and can have significant impacts on the performance.

5.3 Evolution Layer

The implementation of the evolution layer encompasses components that have to do with the genetic process. All components presented here are directly based on the theoretical description of the layer in the previous chapter.

5.3.1 Binary Chromosome

There are many ways to implement a genetic chromosome. A common and simple variant is to use strings to store the data as single bytes. However, because

strings store the data as characters, they make it more difficult to access it in greater chunks. This can hamper efficiency, especially on a 64-Bit CPU that could potentially process eight bytes at once. Although there are ways to mitigate this problem, it as a cleaner solution to implement a proper `BinaryChromosome` class that uses register-sized chunks to store the binary data.

Listing 5.3: Abridged interface of the `BinaryChromosome` class

```
struct BinaryChromosome
{
  // type definitions
  using word_type = uint64;
  // (...)

  //  constructors and destructor
  BinaryChromosome();
  template <typename RNG> BinaryChromosome(size_t size, RNG& rng);

  // comparison operators
  bool operator == (const BinaryChromosome& rhs) const;
  bool operator < (const BinaryChromosome& rhs) const;

  // accessors
  size_t size() const;
  void setFitness(double fitness);
  double fitness() const;
  word_type getWord(size_t begin, size_t count) const;
  void setWord(size_t begin, size_t count, word_type word);

  // iterators
  // (...)

  // genetic operator helpers
  void mutate(size_t index);
  void swap_range(BinaryChromosome& rhs, size_t first, size_t last);

private:

  // members
  std::vector<word_type> _words;
  size_t _size;
  double _fitness;
};
```

Listing 5.3 shows an abridged interface of the `BinaryChromosome` class. Note how the data gets stored word-wise in a dynamic data structure. An alternative variant would be to use an array to create a fixed-size chromosome. The chromosome can either be default-constructed, in which case its data gets zeroed out, or it can be created with random data using the constructor template. The template parameter specifies the type of the *random number generator (RNG)* that is used to create the data. This can be useful in certain applications where higher quality RNGs are required.

Note that the chromsome does not include the functionality to encode or decode data. The reason for this is that the encoding depends on a particular problem and including it here would mean that the chromosome would be coupled to the PE approach. Instead, the chromosome provides functions to access the raw data. These functions use bit-masks and bit-shift operations to extract or write specific

ranges of the content. As was mentioned in 4.2.1, chromosomes are often implemented as value types in regards to their content and do not allow manipulation of the data. But because the PE approach needs to encode the parameters into the chromsomes, the class also offers a `setWord` function in addition to the `getWord` function to read the data.

Aside from being able to set the fitness of the chromosome, it also provides helper functions for the genetic operators. They allow to perform in-place crossover and mutation operations, without having to use the accessor functions to read and write the data manually.

Finally, the `BinaryChromosome` also allows the usage of *iterators* to access the raw data. This functionality is primarily useful for the implementation of some of the genetic operators.

5.3.2 Population

The `Population` class represents a generation of chromosomes.

Listing 5.4: Abridged interface of the `Population` class

```
template <typename ChromosomeType>
struct Population
{
  // type definitions
  using chromosome_type = ChromosomeType;
  // (...)

  // setters and getters
  void setGeneration(size_t generation);
  void setMaxPopulationSize(size_t maxSize);
  size_t maxPopulationSize() const;
  size_t populationSize() const;
  size_t generation() const;

  // chromosomes
  template <typename... Params> void add(Params&&... params);
  void clear();
  bool contains(const chromosome_type& chromosome) const;
  void erase(iterator it, iterator end);
  void remove(const chromosome_type& chromosome);

  // iterators
  // (...)

  // index operator
  chromosome_type& operator [] (size_t index);

private:

  // members
  size_t _maxPopulationSize;
  size_t _currentGeneration;
  std::vector<chromosome_type> _chromosomes;
};
```

Listing 5.4 shows an abridged interface of the `Population` class. Note that the class does not rely on a specific chromosome type, but instead offers a template

parameter that can be used to specify which chromosome variant is used. This allows the population to work with arbitrary chromosome implementations, instead of being restricted to the `BinaryChromosome`.

Because the class is essentially a container for chromosomes, its functionality is pretty straightforward. Aside from providing setters, getters, iterators and an index operator to access the individual chromosomes, also offers functions to add new chromosomes to the population. The `add` function is templated, to allow the in-place construction of new chromosomes, using a C++ technique called *perfect forwarding*.

5.3.3 Genetic Operators

The genetic operators will only be described very briefly, because they are direct implementations of the corresponding methods described in 4.2.2.

Selection

As was stated before, there are several different algorithms for chromosome selection. Each selection class thereby represents one of those methods.

Listing 5.5: Abridged interface of the `RouletteSelection` class

```
1  template <typename PopulationType>
2  struct RouletteSelection
3  {
4    // accessors
5    void setPopulation(PopulationType& population)
6
7    // select chromosomes
8    template <typename RNG>
9    chromosome_type select(RNG& rng) const;
10   template <typename RNG>
11   std::vector<chromosome_type> select(size_t count, RNG& rng) const;
12
13 private:
14
15   // members
16   const PopulationType* _populationPtr;
17 };
```

Listing 5.5 shows the abridged interface of the `RouletteSelection` class as an example for a selection implementation. The class is templated to be usable with arbitrary population types. Other selection methods have a similar interface, although they might provide additional functions for configuration. For example, the `LinearRankSelection` allows to set a *selective pressure*, while the `TournamentSelection` can be configured to use a certain number of competitors.

To select chromosomes the user has to set the corresponding population and use one of the `select` functions. Selection algorithms usually involve the use of random numbers and as with the *BinaryChromosome*, the functions are templated so that the user can specify the RNG type.

Crossover

The `Crossover` class represents the crossover part of the genetic process. It is used to create new chromosomes.

Listing 5.6: Abridged interface of the `Crossover` class

```
1  struct Crossover
2  {
3    // configuration
4    void setMinXops(size_t minXop);
5    void setMaxXops(size_t maxXop);
6    void setPossibleXops(const std::vector<size_t>& xops);
7    template <size_t N> void setPossibleXops(size_t(&xops)[N]);
8    void setFixedXops(const std::vector<size_t>& xops);
9    template <size_t N> void setFixedXops(size_t(&xops)[N]);
10   void setXoPropability(double p);
11   void setToggleXopsProbability(double p);
12   // (...)
13
14   // crossover
15   template <typename ChromosomeType, typename RNG>
16   auto createChildren(ChromosomeType parent1, ChromosomeType parent2,RNG& rng)
17     -> std::pair<ChromosomeType, ChromosomeType>;
18
19 private:
20
21   // helper functions
22   template <typename RNG>
23   void determineXops(size_t chromosomeSize, RNG& rng);
24
25   // type definitions
26   typedef std::pair<size_t, size_t> XoRange_t;
27
28   // members
29   //(...)
30   std::vector<XoRange_t> _xoRanges;
31 };
```

Listing 5.6 shows an abridged version of the `Crossover` class interface. With smaller problems it is often sufficient to use a *single point* or *two point* crossover with either random or fixed *crossover points (xops)*. To offer some flexibility, the `Crossover` class can be configured with via a wide variety of options. The number of crossover points is a random value between the minimal and maximal crossover settings. Additionally, the user can configure a probability that determines whether the crossover point is actually used or not. Although there are many different ways to apply crossover, the evaluation of the Hybrid Approach suggests that the actual method used does not have a significant impact on the effectiveness of the optimization, unless way too many or way too few crossover points are used in regards to the chromosome size.

To create a new pair of children based on two parents, the user can call the `createChilden` function, which is again templated to work with arbitrary chromosome types. As with the selection, the used RNG type can also be specified as a template argument.

Listing 5.7: Definition of the of the `createChildren` function in the `Crossover` class

```
template <typename ChromosomeType, typename RNG>
auto Crossover::createChildren(ChromosomeType parent1, ChromosomeType parent2,
  RNG& rng) -> std::pair<ChromosomeType, ChromosomeType>
{
  // create (new) crossover points (xops)
  determineXops(parent1.size(), rng);

  // create the offsprings
  for (const std::pair<size_t, size_t> xo : _xoRanges)
  {
    parent1.swap_range(parent2, xo.first, xo.second);
  }

  // return the offsprings
  return std::make_pair(std::move(parent1), std::move(parent2));
}
```

Listing 5.7 shows how the children are created. The internal function `determineXops` creates a set of crossover ranges based on the current configuration. The determined crossover ranges are then swapped, using the helper function of the chromosome.

Mutation

The `Mutation` class represents the mutation part of the genetic process. It is used to introduce some random variation to the chromosomes to keep a generation from becoming too homogeneous.

Listing 5.8: Abridged interface of the of the `Mutation` class

```
struct Mutation
{
  // configuration
  void setMinMups(size_t minMup);
  void setMaxMups(size_t maxMup);
  void setMutationPropability(double p);

  // mutate
  template <typename ChromosomeType, typename RNG>
  ChromosomeType mutate(ChromosomeType chromosome, RNG& rng) const;

private:

  // members
  // (...)
};
```

Listing 5.8 shows an abridged version of the `Mutation` class. There a many different variants to implement the mutation algorithm. Like the `Crossover` class, the `Mutation` class provides a way to configure how the mutation is performed. The user can then use `mutate` to mutate a chromosome. The function will determine the number of *mutation points (mups)* based on the configuration and flip the corresponding bits accordingly. Analogous to he `Crossover` class, the `Mutation` can be used with arbitrary chromosome and RNG types.

5.3.4 Genetic Solver

The `GeneticSolver` class represents an implementation of the Genetic Process.

Listing 5.9: Abridged interface of the `GeneticSolver` class

```
template <typename S, typename C, typename M, typename RNG = std::mt19937_64>
struct GeneticSolver
{
  // type definitions
  using selection_type = S;
  using crossover_type = C;
  using mutation_type = M;
  using rng_type = RNG;

  // accessors
  void setPopulationSize(size_t size);
  size_t populationSize() const;
  void setNumberOfGenerations(size_t num);
  size_t numberOfGenerations() const;

  mutation_type& mutation();
  crossover_type& crossover();
  selection_type& selection();

  // solve problem
  template <typename DescType>
  typename DescType::chromosome_type solve(const DescType& desc);

private:

  // initial population
  template <typename DescType>
  auto createInitialPopulation(const DescType& desc)
    -> typename DescType::population_type;

  // members
  rng_type _rng;
  selection_type _selection;
  crossover_type _crossover;
  mutation_type _mutation;
  size_t _numberOfGenerations;
  size_t _populationSize;
};
```

Listing 5.9 shows an abridged version of the `GeneticSolver` interface. Following the paradigm of generic design, the component is a class template, which allows all essential parts of the Genetic Process to be specific via template arguments. This allows the `GeneticSolver` to be used with arbitrary optimization problems, which are not only not limited to the PE approach, but also not limited to regression problems in general.

The user can create a `GeneticSolver` object and use the corresponding accessors to configure the specified Selection, Crossover and Mutation methods, as well as the size of each population and the maximal generations the solver should evolve before termination.

After the solver has been configured, the user can start the Genetic Process by calling the `solve` function. This function is invoked with an arbitrary *genetic description* object, that contains the functionality needed to evaluate the chro-

mosomes. In the PE approach, this is the `NonlinearRegression_Description` type.

Listing 5.10: Definition of the `solve` function of the `GeneticSolver` class

```
template <typename S, typename C, typename M, typename RNG>
template <typename DescType>
auto GeneticSolver<S, C, M, RNG>::solve(const DescType& desc)
  -> typename DescType::chromosome_type
{
  // type definitions
  using desc_type = DescType;
  using chromosome_type = typename DescType::chromosome_type;
  using population_type = typename DescType::population_type;

  // initialization
  population_type currenPopulation = createInitialPopulation(desc);
  chromosome_type fittestChromosome = desc.createRandomChromosome(_rng);

  // create the generations
  for (size_t i = 1; i <= _numberOfGenerations; ++i)
  {
    // create a new population
    population_type tmp(_populationSize, i);

    // transfer the very best chromosome to the next generation (unchanged)
    tmp.add(*currenPopulation.begin());

    // create new chromosomes with crossover & mutation
    _selection.setPopulation(currenPopulation);
    while (tmp.populationSize() < _populationSize)
    {
      std::vector<chromosome_type> selected = selection.select(2, _rng);
      std::pair<chromosome_type, chromosome_type> offsprings =
        _crossover.createChildren(std::move(selected[0]),
                    std::move(selected[1]), _rng);
      tmp.add(_mutation.mutate(std::move(offsprings.first), _rng));
      tmp.add(_mutation.mutate(std::move(offsprings.second), _rng));
    }
    tmp.erase(std::begin(tmp) + _populationSize, std::end(tmp));

    // evaluate all chromosomes in the population
    desc.evaluatePopulation(tmp);

    // sort the chromosomes by fitness
    std::sort(std::begin(tmp), std::end(tmp),
      is_fitter_than<chromosome_type>());

    currenPopulation = std::move(tmp);
    fittestChromosome = *currenPopulation.begin();
  }

  return fittestChromosome;
}
```

Listing 5.10 shows the definition of the `solve` function. Note that the types used for the chromosomes and the population are provided by the description. This allows the user to have additional flexibility when creating a description for an individual problem. Otherwise the function is a straightforward implementation of the generic process shown in figure 2.1.

5.3.5 Nonlinear Regression Description

Because genetic algorithms are supposed to be agnostic about what exactly they are optimizing, it is necessary to introduce a separate component that encapsulates the necessary functionality and information that is needed by the genetic process, but that is specific to a particular problem. This is achieved by introducing the concept of *genetic descriptions*. As the name suggests, a description class describes a genetic optimization problem and is used by the `GeneticSolver` to evaluate the chromosomes.

The `NonlinearRegression_Description` represents the description of a nonlinear regression problem.

Listing 5.11: Abridged interface of the `NonlinearRegression_Description` class

```
template <typename Traits, typename NumericSolver>
struct NonlinearRegression_Description
  : public Regression<Traits>, public RegressionParameterConfig<Traits>
{
  // static constants
  static const size_t NumVariables = Traits::NumVariables;
  static const size_t NumParameters = Traits::NumParameters;

  // type definitions
  using traits_type = Traits;
  using fp_type = typename Traits::fp_type;
  using chromosome_type = typename Traits::chromosome_type;
  using population_type = typename Traits::population_type;
  using result_type = typename Traits::result_type;
  using model_type = typename Traits::model_type;
  using solver_type = NumericSolver;
  using encoding_type = encoding_type;

  // constructors and destructor
  NonlinearRegression_Description(model_type model);

  // evaluation
  void evaluateChromosome(chromosome_type& chromosome,
    result_type* result) const;
  void evaluatePopulation(population_type& population) const;

  private:

  // decode/encode
  void decode(std::vector<fp_type>& parameters,
    std::vector<fp_type>& diffSums, population_type& population) const;
  void encode(population_type& population,
    std::vector<fp_type>& parameters, std::vector<fp_type>& diffSums) const;

  // solve
  void solve(int32 populationSize,
    std::vector<fp_type>& parameters, std::vector<fp_type>& diffSums) const;

  model_type _model;
};
```

Listing 5.11 shows and abridged version of the `NonlinearRegression_Description` interface. The class is templated with two type parameters. The `Traits` type is a helper class that contains all the types that are necessary for the genetic process. This is where the user can specify, among other things, which chromosome and

population types are used in the genetic process. In particular, the traits type allows the specification of the type of encoding that is used to read and store the chromosome data.

The second template parameter `NumericSolver`, specifies which nonlinear regression algorithm is used. Not only allows the parameter to select the particular NRA method, such GNA, LMA or MLA algorithms, but also to choose between a CPU and AMP implementation of the solver.

The traits type also contains the `fp_type` (floating point type) that is used within the nonlinear regression evaluation. Obvious choices are `float` or `double`, but the generic design would also allow any other compatible floating point type to be used, including custom types with higher precision. The choice of the `fp_type` can have a major impact to both accuracy and performance of the PE approach.

Finally, the description also holds the nonlinear regression model. However, because it is first used in Evaluation Layer, it is described in more detail when discussing the types regarding the regression implementation.

The central functionality of the `NonlinearRegression_Description` class is the ability to decode and evaluate chromosomes.

Listing 5.12: Definition of the `evaluatePopulation` function in the `NonlinearRegression_Description` class

```
1  template <typename Traits, typename NumericSolver>
2  void NonlinearRegression_Description<Traits,
3    NumericSolver>::evaluatePopulation(population_type& population) const
4  {
5    const int32 populationSize =
6      static_cast<int32>(population.populationSize());
7    std::vector<fp_type> parameters;
8    std::vector<fp_type> diffSums;
9    parameters.resize(populationSize * _numParameters);
10   diffSums.resize(populationSize);
11
12   // decode all chromsomes
13   decode(parameters, diffSums, population);
14
15   // solve regression for each proposed set of parameters
16   solve(populationSize, parameters, diffSums);
17
18   // encode all chrosomes
19   encode(population, parameters, diffSums);
20 }
```

Listing 5.12 shows the definition of the `evaluatePopulation` function. First, all chromosomes are decoded and the contained parameters written into a dedicated container. Note that the parameters are stored sequentially in single data structure. This means that in the case of a regression model with three parameters, every consecutive three values belong to a single chromosome. The parameters are then passed to the `solve` function, which calls the `NumericSolver` to use the parameters as starting values in the corresponding NRA algorithm. The pa-

rameter container in the `solve` function is an in-out parameter and contains the result parameters of the NRA methods when the function returns. The parameters are then encoded into the chromosomes of the population, which is essentially the reverse operation of the `decode` function. Although it is not explicitly shown here, it should be noted that the `encode` and `decode` functions utilize OpenMP to parallelize their corresponding operations for the whole population.

5.3.6 Encoding

As stated in the previous chapter, the choice of encoding plays a major role in the effectiveness of genetic algorithms in general and the PE approach in particular. To prevent having to implement a new description type for every variation, the encodings are implemented as their own helper types, which can be specified to the `NonlinearRegression_Description` via the `Traits` parameter.

Listing 5.13: Implementation of the `DiscretizedEncoding` class

```
template <typename FP>
struct DiscretizedEncoding
{
  static FP decode(uint64_t word, FP minValue, FP maxValue, size_t binarySize)
  {
    return minValue + (maxValue - minValue) * (static_cast<FP>(word)
    / (~static_cast<uint64>(0) >> (64 - binarySize)));
  }

  static uint64_t encode(FP value, FP minValue, FP maxValue,size_t binarySize)
  {
    const uint64 maxPossibleValue = (binarySize == 64) ?
      (std::numeric_limits<uint64>::max()) :
      (static_cast<uint64>(1) << binarySize);
    const FP factor = maxPossibleValue / (maxValue - minValue);

    return static_cast<uint64>(factor * (value - minValue));
  }
};
```

Listing 5.13 shows the definition for the `DiscretizedEncoding` class. This is a straight up implementation of the definition that was given in 4.2.3, with the only difference that it uses 64 bit integers as words as opposed to 32 bit. Note that this variant provides the option to specify a binary size, because this is a requirement for the logarithmic encoding. It also allows the the floating point type to be specified using the `FP` template parameter.

Listing 5.14: Implementation of the `LogarithmicEncoding` class

```
template <typename FP>
struct LogarithmicEncoding
{
  static FP decode(uint64_t word, FP minValue, FP maxValue)
  {
    auto value = word & 0x7FFFFFFFFFFFFFFF;
    auto sign = word & 0x8000000000000000;

    return sign | std::pow(static_cast<FP>(10.0),
      DiscretizedEncoding<FP>::decode(value, minValue, maxValue, 63));
  }

  static uint64_t encode(FP value, FP minValue, FP maxValue)
  {
    auto word = DiscretizedEncoding<FP>::encode(std::log10(std::abs(value)),
    minValue, maxValue, 63);
    auto sign = value & 0x8000000000000000;
    word |= sign;

    return word;
  }
};
```

Listing 5.14 shows the definition for the `LogarithmicEncoding` class. This again is a direct implementation of the formal definition. Similar to what was shown in 4.2.3, the logarithmic encoding is implemented by using the discretized encoding with an additional logarithmic transformation. It is important to note the bit operations on the floating point values are based on the assumption that the `FP` type complies to the IEEE 754 floating point standard. This is verified at compile time by a `static_assert` instruction; which is omitted here to improve readability.

5.4 Evaluation Layer

The implementation of the evaluation layer encompasses all components related to nonlinear regression analysis. Contrary to the evolution layer, some crucial components cannot directly be derived from what was presented in chapter 4. This has primarily to do with the intricacies of implementing mathematical concepts. Although the regression model function and its derivatives are comparatively simple to define and handle from theoretical point of view, they pose a few challenges to an implementation in software.

5.4.1 Regression Model

One challenging aspect of the PE approach is the question of how to make the regression model available to the NRA method. Not only does the computation of the residual require access to the regression function, many nonlinear regression algorithms are so called *derivative-based* methods, meaning that they also require the partial derivatives of the regression function for every parameter θ_j. In case of the LMA and MLA methods used in this implementation, they are required to update the Jacobian matrix in every iteration.

Usually, the regression model could be implemented by using a model interface. A user could then implement that interface for every model he or she wants to use. An even simpler solution would be to use an array of function pointers, which would refer to both the regression function and its derivatives. However, as was mentioned in 5.2, AMP-restricted code does not allow virtual functions or function pointers in general, because an accelerator has no concept of a call stack that can be used to invoke functions in the traditional sense. Instead, all functions have to be known at compile time.

One possible solution for this problem is to use a *tuple* data structure. In generic programming, a tuple is a static container for elements of different types. Although the C++ standard library contains a tuple class in the form of `std::tuple`, it cannot be used because its implementation is not AMP-restricted. Even though properly implementing a tuple is not a trivial matter, a simplified version can be written with a comparatively little amount of code.

Listing 5.15: Abridged interface of the `Tuple` class

```
template <typename...>
struct Tuple;

template <>
struct Tuple<>
{
  Tuple() = default;
};

template <typename T, typename... Ts>
struct Tuple<T, Ts...> : Tuple<Ts...>
{
  using base_type = Tuple<Ts...>;

  Tuple(T&& f) restrict(cpu, amp);
  template <typename = std::enable_if_t<sizeof...(Ts) >= 1>>
  Tuple(T&& f, Ts&&... fs) restrict(cpu, amp);

  template <int Idx>
  auto& get() restrict(cpu, amp);

  __declspec(align(4)) T _t;

private:

  template <int, typename>
  struct Tuple_Element;

  template <int Idx, typename U, typename... Us >
  struct Tuple_Element<Idx, Tuple<U, Us...>>
    : Tuple_Element<Idx-1, Tuple<Us...>> {};

  template <typename U, typename... Us>
  struct Tuple_Element<0, Tuple<U, Us...>>
  {
    using type = Tuple<U, Us...>;
  };
};
```

Listing 5.15 shows an abridged interface of the `Tuple` class. It uses several tech-

niques of template meta-programming to handle the individual elements of potentially different types. Because the actual implementation is rather technical and specific to C++, it will be omitted here. The important aspect to take away is that the tuple can be constructed with a templated constructor and that the elements can be accessed by using the `get` function along with the corresponding index of the element.

Listing 5.16: Construction of a regression model by using the `tuple`

```
// create model
auto f = [](fp_type t_0, fp_type t_1, fp_type x_0) restrict(cpu, amp)
{
  return t_0 * pow(x_0, t_1);
};

auto d0 = [](fp_type /*t_0*/, fp_type t_1, fp_type x_0) restrict(cpu, amp)
{
  return pow(x_0, t_1);
};

auto d1 = [](fp_type t_0, fp_type t_1, fp_type x_0) restrict(cpu, amp)
{
  return t_0 * pow(x_0, t_1) * log(x_0);
};

auto model = make_amp_tuple(std::move(f), std::move(d0), std::move(d1));
```

Listing 5.16 shows how a tuple can be used to create a regression model. The example defines the regression function `f` as well as its partial derivatives as function objects, which are created by using lambda functions. These function objects are then used to construct a tuple object. Note that the lambda functions are restricted to both AMP and CPU usage. This example is taken from the testing environment that is used in the evaluation of the PE approach, which is presented in chapter 6.

With the usage of a generic computer algebra system, the derivatives could even be determined at compile time, so that the user would only have to define the regression function to create a regression model.

5.4.2 Nonlinear Regression Solver

The main part of the regression layer is the component that represents a nonlinear regression algorithm. For this, the PE approach introduces the concept of a *nonlinear regression solver*.

Listing 5.17 shows an abridged interface of the `NonlinearRegression_Amp` class. This class represents the AMP version of an NRA solver that provides implementations for both the LMA and MLA methods. Which method is used can be – among other things – specified by the corresponding entry in the traits type, which is passed as a template argument to the class. It should be noted that a lot of details are omitted in this description and that the actual declaration contains several definitions of aliases and helper types that are used in the implementation.

Listing 5.17: Abridged interface of the `NonlinearRegression_Amp` class

```
template <typename Traits>
struct NonlinearRegression_Amp
{
  static const int TileSize = 32;

  // member functions
  template <typename Model>
  static void solve(int populationSize, std::vector<fp_type>& diffSums,
            std::vector<fp_type>& parameters, std::vector<fp_type>& ys,
            std::vector<fp_type>& xs, Model& model);

private:

  // (...)

  static void solve_step0_init(tiled_index& tidx,
                array_view<fp_type, 1>& diffSums,
                array_view<fp_type, 2>& parameters,
                TileData& td) restrict(amp);

  template <typename Model>
  static void solve_step1_update(Model& model,
                  ys_container& ys,
                  xs_container& xs,
                  TileData& td) restrict(amp);

  static void solve_step2_solve(tiled_index& tidx,
                  int numDataPoints,
                  TileData& td) restrict(amp);

  template <typename Model>
  static void solve_step3_propose(Model& model,
                   ys_container& ys,
                   xs_container& xs,
                   TileData& td) restrict(amp);

  static void solve_step4_exit(tiled_index& tidx,
                 TileData& td,
                 array_view<fp_type, 1>& diffSums,
                 array_view<fp_type, 2>& parameters) restrict(amp);
};
```

As the name suggests, a NRA solver class can solve a particular nonlinear regression problem. The problem is passed to the `solve` function along with the data points as well as the starting values that were decoded from the chromosomes in the `NonlinearRegression_Description`. The class does not contain any data members, because an AMP-restricted lambda is not allowed to capture the `this` pointer to the surrounding class object. For this reason, all functions are static and the data has to be passed from function to function. To reduce the number of parameters, some of the data is encapsulated in the the `GlobalData` and `TileData` helper types.

It is important to note that the layout of the data can have a big impact on the overall performance of parallelized code. In CPU code, collective data is often encapsulated in structures and if there are several instances they can be stored in an array. This approach is also referred to as an *array of structures (AoS)*. In the PE approach, this would be equivalent to encapsulating all the data relevant to the

evaluation of a single chromosome into a structure, and then creating an array of those structures to hold the data of the complete population. The AoS approach is beneficial for CPUs, because each CPU core has its own cache and only runs a single thread at a time. This means that a large amount, if not all, of the data relevant to the thread can conveniently be read from memory and stored in the cache.

GPUs on the other hand are designed to execute a number of threads in parallel. Nvidia chips for example consist of several so called *streaming multiprocessors (SM)* each of which can run 32 threads at once. For this reason, consecutive data in the global memory is accessed in bulks of $32 \cdot 4$ bytes. This means that with an optimal data layout, the data for all threads may be fetched with a single read to the global memory. This can have a significant impact on the performance, because global memory access on a GPU is comparatively slow. Therefore, well designed GPU code should always try to minimize global memory accesses as much as possible. For this reason, the data in the AMP code is stored in different arrays of the specified floating point type. To make passing these arrays easier, they can be bundled in a single instance of a structure. In contrast to the AoS approach, this strategy is also known as *structure of arrays (SoA)*.

Due to its complexity, it is not feasible to show the entire implementation of the `NonlinearRegression_Amp` class. Instead, listing 5.18 shows an abridged implementation of the `solve` function, to offer a rough overview of the calling hierarchy. Variables denoted with the prefix `_v` refer to array views of the corresponding data containers. Because AMP does not allow the dynamic allocation of memory, all working containers for temporary vectors and matrices have to be created on the CPU and then captured by the kernel. All this data is stored in the `gd` variable of the `GlobalData` type. Within each kernel, a variable `td` of the helper type `TileData` is used to hold references to all data relevant to the current tile.

The first kernel initializes the solver by copying the parameters to the working containers. The main loop then executes the actual NRA algorithm. Aside from a few matrix operations, each step mostly delegates the computation to the corresponding components of the computation layer. Note that the loop runs for a fixed amount of 50 iterations and currently does not use any kind of convergence criterion for termination. The main reason for this is to provide comparable running times in the evaluation of the approach. Although using a convergence check might improve the performance, it might also skew the individual measurements because a "lucky run" might terminate much faster than another. Additionally, the PE approach does not necessarily need each evaluation to achieve maximal convergence.

Note that the loop is not contained within the kernel, but is surrounding it. This to ensure that the kernel does not take too much time. Because users usually do not have a dedicated graphics card for GPGPU computing, they are often using the same GPU for data processing that is also responsible for rendering the desktop

environment of the operating system. Because a GPU cannot switch the currently executed code as easily as a CPU can, an AMP kernel blocks the GPU until the kernel hast returned. Because it is not possible to tell whether a piece of code intentionally blocks the GPU, many modern operating systems have protection mechanisms in place to prevent a lockup of the desktop environment. Windows in particular uses a feature called *Timeout Detection & Recovery (TDR)*, which restarts the driver if it is not responding for more than 1.5 seconds. Although the TDR limit can be increased, it is recommended to instead rearrange the code in such a way that every kernel execution does not exceed the set time limit.

In addition to the AMP solver, the implementation also contains the a corresponding solver for the CPU in the form of the `NonlinearRegression_CPU` class. The CPU variant uses an AoS approach and utilizes OpenMP to evaluate several chromosomes at once. However, because it otherwise has a very similar general structure as the AMP solver, it is omitted here.

Listing 5.18: Abridged implementation of the `solve` function in the `NonlinearRegression_Amp` class

```
1  template <typename Traits>
2  template <typename Model>
3  void NonlinearRegression_Amp<Traits>::solve(int populationSize,
4    std::vector<fp_type>& diffSums, std::vector<fp_type>& parameters,
5    std::vector<fp_type>& ys, std::vector<fp_type>& xs, Model& model)
6  {
7    (...)
8    GlobalData gd(workIdx, slideIdx, lambda, diffSum, theta, r, J, Tmp, Tmp2);
9
10   parallel_for_each(ext, [&](tiled_index tidx) restrict(amp)
11   {
12     TileData td(tidx, gd);
13     solve_step0_init(tidx, v_diffSums, v_parameters, td);
14   });
15
16   acc_view.wait();
17
18   // main loop
19   for (int h = 0; h < 50; ++h)
20   {
21     parallel_for_each(ext, [&](tiled_index tidx) restrict(amp)
22     {
23       TileData td(tidx, gd);
24       solve_step1_update(model, v_ys, v_xs, td);
25       solve_step2_solve(tidx, numDataPoints, td);
26       solve_step3_propose(model, v_ys, v_xs, v_parametersConfig, td);
27     });
28   }
29
30   // copy data from working temporaries
31   parallel_for_each(ext, [&](tiled_index tidx) restrict(amp)
32   {
33     TileData td(tidx, gd);
34     solve_step4_exit(tidx, td, v_diffSums, v_parameters);
35   });
36
37   v_diffSums.synchronize();
38   v_parameters.synchronize();
39 }
```

5.4.3 Linear System Solver SVD

In step 2, the NRA method has to perform the update of the parameter vector $\boldsymbol{\theta}$. As was shown in 4.3.1, this requires to solve a difficult system of equations. Here, this functionality is implemented in its own type instead of in the NRA algorithm, to make it reusable in different contexts.

The `LinearSystemSolverSVD` is a functor type that can solve a system of linear equations of the form $\boldsymbol{Ax} = \boldsymbol{b}$, by using the singular value decomposition of $\boldsymbol{A}$. It is used in the parameter update of the NRA algorithm.

Listing 5.19: Implementation of the calling operator of the `LinearSystemSolverSVD` class

```
template <int M, int TS, typename FP>
void operator () (tiled_index<TS>& tidx, TiledContainer<TS, FP, 3>& Tmp,
  TiledMatrix<TS, FP>& A, TiledVector<TS, FP>& x,
  TiledVector<TS, FP>& b) restrict(amp)
{
  constexpr int TileStaticMemory = 32768;
  constexpr int TileStaticEntries = M * M * TS;
  constexpr int TileStaticMatrix = TileStaticEntries * sizeof(FP);
  constexpr bool TileStaticSigma = TileStaticMatrix <= TileStaticMemory;
  constexpr bool TileStaticU = 2 * TileStaticMatrix <= TileStaticMemory;
  constexpr bool TileStaticV = 3 * TileStaticMatrix <= TileStaticMemory;

  // create temporary matrices and vectors
  tile_static FP U_[TileStaticU ? TileStaticEntries : 1];
  tile_static FP Sigma_[TileStaticSigma ? TileStaticEntries : 1];
  tile_static FP V_[TileStaticV ? TileStaticEntries : 1];

  TiledMatrix<TS, FP> U = TileStaticU ?
    TiledMatrix<TS, FP>(tidx.local[0], extent<3>(M, M, TS), U_) :
    Tmp.getSection(1);
  TiledMatrix<TS, FP> Sigma = TileStaticSigma ?
    TiledMatrix<TS, FP>(tidx.local[0], extent<3>(M, M, TS), Sigma_) :
    Tmp.getSection(2);
  TiledMatrix<TS, FP> V = TileStaticV ?
    TiledMatrix<TS, FP>(tidx.local[0], extent<3>(M, M, TS), V_) :
    Tmp.getSection(3);

  // calculate: A = U * Sigma * VT
  JacobiRotationSVD<M, TS>()(A, U, Sigma, V);

  // calculate: Sigma^+
  for (int i = 0, j = 0; i < rows(Sigma); ++i, ++j) {
    get(Sigma, i, j) = rcp(get(Sigma, i, j));
  }
  transpose(Sigma);

  // calculate pesudo inverse of A
  transpose(U);
  mul(V, Sigma, A);
  mul(A, U, V);

  // calculate x
  mul(V, b, x);
}
```

Listing 5.19 shows the definition of the AMP version of the `LinearSystemSolverSVD` calling operator, which essentially is a straight up implementation of what was presented in 4.3.1. The template argument `M` denotes the number of regression parameters, while `TS` is the tile size. An important detail is the automatic use

of `tile_static` memory. The computation of the SVD requires several matrices, whose size in the case of the LMA and MLA is entirely dependent on the number of parameters. Because the number of parameters is known at compile time, the implementation can statically determine how many of the matrices can be stored in the tile static memory. The `TiledVector` and `TiledMatrix` classes are based on the `array_view` type and can therefore either use the tile static memory if enough is available or use the temporary matrices in the global memory as a fallback.

The computation of the SVD is then delegated to the `JacobiRotationSVD` class – which is also a functor type – that uses the Jacobi rotation SVD method to decompose a matrix.

5.5 Computation Layer

The implementation of the computation layer consists of all components related to the actual calculation of the NRA algorithm. It includes a lot of implementation details necessary to perform the various operations of the PE approach. For the sake of brevity, only the most important components are covered here. Most notably, it is not shown how matrix and vector operations are implemented, because they are mostly straight up versions of common algorithms. Instead, this section focuses on the evaluation of the regression model and the computation of the SVD.

5.5.1 Model Evaluator

Section 5.4.1 of the evaluation layer showed how the regression model can be represented as a tuple of lambda functions. Calling the stored functions requires another heavy use of template meta-programming.

Listing 5.20 shows the implementation of the `ModelEvaluator` helper class, which is called from the NRA algorithm to evaluate the model function. To perform the evaluation, the regression function has to be called with the corresponding arguments from the data container $\boldsymbol{x}_k$ and the parameter vector $\boldsymbol{\theta}$. To do this, the evaluator constructs an integer parameter pack that is passed to an internal function. The deduced parameter pack can then be expanded to statically construct the call for the correct number of parameters and predictor variables. The `DerivativesEvaluator`, which is called from the NRA algorithm to update the Jacobian, is implemented using the same general technique and is omitted here.

Listing 5.20: Implementation of the `ModelEvaluator` class

```
template <int NumVar, int NumPar>
struct ModelEvaluator
{
  template <int TS, typename Model, typename FP>
  auto operator () (Model& model, vector_view<const FP>& x,
    TiledVector<TS, FP>& theta, int workIdx) restrict(amp)
  {
    return (*this)(model, x, theta, workIdx,
      std::make_integer_sequence<int, NumVar>(),
      std::make_integer_sequence<int, NumPar>());
  }

private:

  template <int TS, typename Model, typename FP, int... VarIdxs, int... ParIdxs>
  auto operator () (Model& model, vector_view<const FP>& x,
    TiledVector<TS, F>& theta, int workIdx,
    std::integer_sequence<int, VarIdxs...>,
    std::integer_sequence<int, ParIdxs...>) restrict(amp)
  {
    return model(theta(workIdx, ParIdxs)..., x(VarIdxs)...);
  }
};
```

Note that although the listing shows the AMP version of the evaluator, the CPU version is implemented in a similar way.

5.5.2 Jacobi Rotation SVD

The `JacobiRotationSVD` is a functor class that computes the SVD decomposition of a matrix. It is for the most part a direct implementation of what was shown in section 4.4.1 and is primarily presented here for the sake of completeness.

Listing 5.21 shows the basic layout of the Jacobi SVD solver. One important difference from the formal algorithm description is the stopping rule. Just like the implementation of the NRA method, the SVD implementation uses a fixed number of iterations instead of a convergence criterion. Again, this is primarily for consistency reasons in the evaluation. Usually, the main loop would only run until the desired accuracy is reached. Another interesting thing to note is that the implementation works on both the CPU and on AMP. This is especially important during the performance evaluation, to ensure that the most computational intensive part of the approach executes the exact same code, regardless of where it runs.

The `rotate` function is omitted, because it is a straight implementation of the algorithm presented in 4.4.1.

Listing 5.21: Implementation of the `JacobiRotationSVD` class

```
template <int M, int TileSize>
struct JacobiRotationSVD
{
  template <typename TMatrix1, typename TMatrix2,
      typename TMatrix3, typename TMatrix4>
  void operator () (TMatrix1& A, TMatrix2& U,
      TMatrix3& Sigma, TMatrix4& V) restrict(cpu, amp)
  {
    // copy A into Sigma, initialize U and V
    for (int i = 0; i < M; ++i) {
      for (int j = 0; j < M; ++j) {
        get(Sigma, i, j) = get(A, i, j);

        if (i == j) {
          get(U, i, j) = 1;
          get(V, i, j) = 1;
        }
        else {
          get(U, i, j) = 0;
          get(V, i, j) = 0;
        }
      }
    }

    // solve
    for (int stop = 0; stop < 50; ++stop) {
      for (int i = 0; i < M-1; ++i) {
        for (int j = i + 1; j < M; ++j) {
          rotate(U, Sigma, V, i, j);
        }
      }
    }

    // zero out all entries in Sigma except diagonal
    for (int i = 0; i < M; ++i) {
      for (int j = 0; j < M; ++j) {
        if (i != j) {
          get(Sigma, i, j) = 0;
        }
      }
    }
  }
};
```

Chapter 6

Evaluation

The last chapter provided an overview of a possible implementation of the PE approach. In particular, it was shown how certain parts can be implemented to use C++ AMP as a means to utilize massive parallelism.

The last remaining goal of the thesis is to use the aforementioned implementation to evaluate the approach. This chapter first describes under which conditions the evaluation was performed. This is done by formulating a set of objectives, which are used as a basis to define the subjects of the evaluation. After providing a description of the testing environment, the remaining part of the chapter presents a summary of the results.

6.1 Setup

The PE approach is a complex interplay between a lot of individual parts and before it is possible to evaluate it, it is necessary to properly define what parts of the approach should be evaluated, which ones should be fixed and under which exact conditions the evaluation should take place.

To do this, the following sections are defining the following terms:

- Objectives: What is the evaluation supposed to achieve?
- Subjects: What is being evaluated?
- Conditions: How is being evaluated?

6.1.1 Objectives

The *objectives* are often formulated in the form of questions which shall be answered by the evaluation. As stated in goal IV., the evaluation shall verify goals I. and II., which were defined as follows:

I. Propose an approach to solve nonlinear regression problems independent of starting values with the following properties:

   (i) It is applicable to a wide variety of practical problems

   (ii) It is suitable for automated use and does not require any additional configuration

   (iii) It is suitable for massive parallelism and fast enough for practical use

II. Propose and examine different strategies to improve the success rate of the approach

Although some of the objectives directly follow from these goals, other are not as obvious and have to be extracted by dissecting each goal and subgoal into several parts.

Before anything else becomes relevant, the most essential aspect of the evaluation is to demonstrate that the approach indeed works in practice and is not just a theoretical construct. After that, it is possible to assess the individual parts to determine whether or not the properties of the approach meet the formulated requirements of goal I..

One of those requirements is the applicability to a wide variety of practical problems. Demonstrating this applicability for the most part hinges on the selection of the test problems that are being used in the evaluation. Although it is generally not possible to cover all types of problems, it should be shown that the test problems at least cover a wide range of typical scenarios.

The subgoal regarding the suitability of the PE approach for automated use consists of several parts and only some of them are relevant for the evaluation. A necessary precondition for automated use of the approach is that this possibility is allowed by its design. If an approach would explicitly require user input at certain points of the execution, automation might not be achievable in principle. An evaluation only becomes relevant in this regard, when an approach offers some kind of configuration. In that case, the evaluation can be used to determine whether or not there is a combination of settings that allows automation in all or at least in most scenarios. This is indeed the case for the PE approach.

Whether or not the approach is suitable for massive parallelism is also mostly contingent upon its design and implementation. In theory, the PE approach should be highly parallelizable, because there are a great number of chromosomes that can be evaluated in the same way, while at the same time being totally independent from one another. What an evaluation can do is to verify whether an approach indeed benefits from parallelization and to determine how it scales with certain types of hardware. Based on the performance measurements of the parallelized

approach, it is then possible to asses whether or not the running times are sufficient for practical purposes.

In addition to goal I., which is mostly concerned about the general properties of the approach, goal II. is about comparing different strategies and techniques for certain parts of it. Because the PE approach is defined and implemented in a generic way, there are a lot of parts that could be examined in greater detail as part of an evaluation. However, due to the complexity of the approach, it is not feasible to meticulously assess every single component. Instead, the evaluation will focus on two aspects that are essential to the approach. The first is the encoding that is used to store the starting parameters in the chromosome and the second one is the choice of the regression method that is used to evaluate them.

These objectives can be summarized in the following questions:

A. Verification: Does the PE approach work in general?

B. Applicability: Is it applicable to a wide variety of practical problems?

C. Practicality: Is it fast enough for practical use?

D. Automation: Is there a configuration that makes it suitable for automated use?

E. Scalability: How does the approach benefit from massive parallelism and how does it scale on different hardware?

F. Analysis: How do the choice of encoding and regression methods affect the success rate?

6.1.2 Subjects

The *subjects* define what quantities of data are collected and what criteria are used to assess them in the context of the evaluation. Subjects usually can be derived from the objectives.

Success Criterion

Both the verification and the applicability objectives require some measure of success. This is not as trivial as it might appear. Although the LMA and MLA methods presented in this thesis are based on the reduction of the least squares error, it is usually not possible to achieve an error of zero, because almost all practical problems contain some forms of random noise or systematic errors in their data points. Since the calculation of the least square error requires the specific values of the corresponding set of parameters – the very thing the regression method is trying to determine – it is usually not possible to know in advance what the error

of an optimal solution would look like. In the evaluation of the Hybrid Approach this problem was circumvented by using generated test problems, which have the advantage that they can be created without any form of noise in such a way that they are guaranteed to be solvable with a least squares error of zero. However, as was stated in the conclusion of that approach, generated problems are also highly artificial. Because one goal of this evaluation is to verify that the PE approach works on a wide range of practical problems, generated data points cannot be used.

Another option in a situation like this is to use reference problems. There are a lot of reference data sets available, which include verified solutions that can be used to test and compare nonlinear regression methods. To comply with objective B., a reference set used in this evaluation would have to be representative of a wide variety of practical problems. One popular set that fulfills this criterion is the *Nonlinear Regression Statistical Reference Data Set*[31] from the *National Institute of Standards and Technology (NIST)*. Although a little outdated, the reference set is still used in many publications in the field of statistics.

As stated in the background information:

> *"In "An Evaluation of Mathematical Software That Solves Nonlinear Least Squares Problems" (ACM Transactions on Mathematical Software, vol. 7, no. 1, March 1981, pages 1-16), Hiebert notes that "testing to find a 'best' code is an all but impossible task and very dependent on the definition of 'best.' " Whatever other criteria are used, the test procedure should certainly attempt to measure the ability of the code to find solutions. But nonlinear least squares regression problems are intrinsically hard, and it is generally possible to find a dataset that will defeat even the most robust codes."*

The description further states:

> *"We have included both generated and "real-world" nonlinear least squares problems of varying levels of difficulty. The generated datasets are designed to challenge specific computations. Real-world data include challenging datasets such as the Thurber problem, and more benign datasets such as Misra1a. The certified values are "best-available" solutions, obtained using 128-bit precision and confirmed by at least two different algorithms and software packages using analytic derivatives."*

This diversity of problems makes the NIST reference set very well suited for the evaluation of the PE approach.

A *success* is hereby defined as a successful convergence to the reference solution of a single problem within a 15% error. To quantify the success of a configuration for the entire test suite, the *number of successes* refers to the amount of test problems that were solved in either more than 50% or 100% of all iterations.

Control

To confirm that the PE approach actually works, it is not sufficient to just examine the success rate. Every empirical experiment needs some form of control to compare the hypothesis (i.e. the approach) to random chance. In the case of the starting value problem, this means that the approach has to be compared to just guessing a set of starting values a certain number of times.

With the PE approach, this can be done by setting the number of generations of the genetic process to zero. This means that the population is evaluated exactly once, without participating in the genetic process. Because the first generation is always randomly generated, this is equivalent of guessing the initial parameters and at the same time guarantees that the control mechanism executes the same evaluation code as the PE approach itself.

Running Time

Another essential objective of the evaluation is the question of practicality, as stated in objective C. The PE approach is computationally very expensive and because it is supposed to be suitable for practical use, it is vital to also analyze its efficiency in addition to its effectiveness.

Performance analyses are often done by just measuring the running times. This poses the potential problem of introducing errors, because on a standard computer software seldom runs isolated on a system and other programs potentially could interfere with the measurement. This is the reason why a proper performance analysis usually involves a profiler. Profilers are tools specially designed to analyze the performance of a program. In many cases this is done by collecting so called *samples*, when the software is actually executing on the CPU. This makes a profiler analysis independent of the running time.

However, although the collection of CPU samples can be useful in many cases, the actual running times are the very things that are essential to answer the question of practicality. Additionally, the PE approach is supposed to utilize heterogeneous computing and potentially does not exclusively run on a CPU.

Instead of using a profiler, the performance will be determined by cautionary measuring the running times for each individual test problem. To mitigate fluctuations due to interference with other software, the measurement will be repeated several times to calculate the arithmetic mean. The approach is considered practical, if it is able to solve many or all problems on common hardware in a reasonable amount of time. Here, common hardware means consumer grade products that are widely available, in contrast to specialist hardware or computer clusters, which usually are a lot more expensive. Finally, what is considered a "reasonable" amount of time is usually subjective and dependent on the context of the problem. However,

it seems sensible that an approach can be considered fast enough if it terminates in under five minutes. This would be short enough to also queue multiple models or problems in under one hour.

6.1.3 Conditions

The *conditions* describe the environment in which the evaluation is performed. This includes both hard- and software, as well as the fixed configurations for those parts of the approach that are not subjects of the evaluation.

Test Suite

In addition to all test problems from the nonlinear regression NIST data set, the evaluation will also use three problems presented by Tvrdik et al[32], which were deemed the most difficult by the author. The problems are *Meyer & Roth Example 4 and 5*, as well as Militky & Meloun Model IV.

In total, the test suite consists of 30 reference problems.

Hardware

The evaluation is performed on the following hardware:

- CPU: Intel i7 4770k [Quadcore at 4.1GHz]
- RAM: 16GB
- Internal GPU: Intel HD Graphics 4600
- Discrete GPU: Nvidia Geforce GTX 1080

All parts are consumer-grade components and fulfill the requirements of objective C.. At the time of writing, the CPU is slightly outdated, while the discrete GPU can be considered "high-end".

Configuration

Every layer of the PE approach contains several components that can be configured in different ways or even entirely replaced with another implementation. Due to the vast amount of possible combinations, it is not feasible to examine all of them. Instead, the evaluation will focus on those parts that are deemed to be the most important. This assessment is primarily based on the results of the Hybrid Approach. The following aspects are part of the dynamic configuration:

- Population Size: 256, 512, 1024, 2048, 4096, 131072 (control only)
- Number of Generations: 5, 15, 25

- Encoding: Discretized, Logarithmic
- NRA Algorithm: Levenberg-Marquardt Algorithm, Modified Levenberg Algorithm

The evaluation of the Hybrid Approach showed that the population size has a significant impact on the success rate. This is not surprising, considering that a greater diversity of possible solutions increases the chances of success. However, since every chromosomes has to be evaluated, increasing the size of each generation also directly influences the running time of the algorithm. Therefore, the population size and the number of generations are important variables for the question of how much effort it takes for the PE approach to produce satisfying results. The most expensive configuration does evaluate $25 \cdot 4096 = 102400$ sets of starting values. The size of the control population is chosen specially to be above that.

The motivation for the different encodings and NRA algorithms were already presented in sections 4.2.3 and 4.3.1 respectively.

All other components and options will be fixed in the evaluation. In particular, the following list consists a of all remaining variables of the PE approach alongside their corresponding settings:

- Genetic Layer:
  - Selection: Roulette Selection
  - Crossover Points: $[5, 10]$
  - Mutation Points: $[5, 10]$
  - Chromosome Binary Size: 64
  - Encoding Range: $\pm[10^{-10}, 10^{10}]$
- Evaluation Layer:
  - Tile Size: 32 (Geforce GTX 1080), 128 (HD 4600)
  - NRA Iterations: 50
- Computation Layer:
  - Decomposition: Jacobi Rotation SVD
  - SVD Iterations: 50
  - Floating-Point Precision: Single, Double (selective)

The choices for the selection, crossover and mutation settings are based on the evaluation results of the Hybrid Approach, where the configuration of the genetic operators had little impact on the success rate. Other settings, such as the binary size for the chromosomes or the max number of iterations of the NRA and SVD

algorithm directly follow from their implementation and the motivations behind them were already stated in chapter 5. The floating-point precision is fixed to single precision for the success rate evaluation, but will be changed to double precision for specific problems, as well as the performance evaluation.

Both the discretized and the logarithmic encoding require to set an encoding range. This is a critical setting, due to the requirement of goal (ii), which stated that the approach must be usable on a wide variety of problems without changing the configuration. It seems reasonable that the range specified above is sufficient for most nonlinear regression problems. However, it still has to be stressed that there might be some problems that require a different configuration of that setting.

Procedure

The evaluation is split into two parts: An examination of the success (rates) and the performance.

The success evaluation is performed on the AMP solver using the Geforce GPU. Because the CPU and the AMP solver use the exact same methods, it is not necessary to compare them or different AMP accelerators in regards to the success rate. To evaluate the success rates, the PE approach is tested on every one of the 30 reference problems of the test suite with all possible 60 combinations of the aforementioned configuration and the control population. To ensure a certain degree of stability in the results, each run is repeated 10 times. This leads to the success rate, which refers to the percentage of successful runs out of the total 10 iterations. Additionally, the success evaluation will also determine the mean running time for every test run.

To assess the potential for parallelization and scalability, additional runs are also performed on a comparatively small test problem (Bennett5) as well as a larger one (Gauss1). Each of these problems will be evaluated using the most successful encoding and regression method of the success rate evaluation. It will be repeated 10 times for every population size. In addition to using the Geforce GPU, all runs are performed on the integrated Intel HD Graphics 4600 GPU of the CPU, as well as the WARP Direct3D emulator that is mentioned in 5.1.3. To provide a point of comparison, the approach also is evaluated using the CPU solver, which does not use C++ AMP at all.

6.2 Results

The complete results of the evaluation are provided in appendix A and B. What follows is a summary of the results.

Success Rates

The success rates for all configurations are given in the tables A.1, A.2, A.3 and A.3. They show the color-coded success rates of the PE approach for every problem of the test suite for every configuration combination, along with the corresponding number of successes.

Best Configurations - Number of Successes (100%)					
Encoding	NRA Method	Population Size / Number of Generations	Precision	Successes [PE Approach]	Successes [Control]
Discretized	LMA	N/A	Single	0/30	0/30
Discretized	MLA	N/A	Single	0/30	0/30
Logarithmic	LMA	4096/15	Single	17/30	0/30
Logarithmic	MLA	4096/15	Single	27/30	1/30
Logarithmic	MLA	4096/15	Double	30/30	3/30

Table 6.1: The highest number of successes for every combination of encoding and NRA method.

Table 6.1 shows the highest number of successes (100%) every combination of encoding and NRA method was able to achieve. What is immediately apparent, is that the discretized encoding was not able to reliably solve a single one of the test problems. In fact, tables A.1 and A.2 show that this encoding was not able to successfully solve any problem of the test suite in any iteration. This is a very surprising result, considering that encoding was successfully used in the evaluation of the Hybrid Approach. To ensure that this is caused by the encoding itself and not an implementation error, the encoding range was decreased for select problems which indeed lead to an increase in the success rate.

The logarithmic encoding on the other hand performed much better, with over half the problems solved when using the LMA solver and all but three problems solved when using the MLA solver. After performing an additional test using double precision, the combination of the logarithmic encoding and MLA method was even able to solve all of the test problems. This confirms that some problems are more sensitive to numerical problems than others.

These results achieve several objectives at once. First and foremost, they verify that the approach works in general – objective A. – because it reliably outperformed the control population, which was only able to solve a few of the lower difficulty problems by random chance. The results also show that the PE approach is applicable to a wide range of different problems – objective B. – by being able to solve every test problem in the test suite. Additionally, these results can be achieved with a single configuration, which satisfies the automation objective D.. However, not all problems require such an expensive configuration. Chart 6.1 shows how the number of successes scale with the population size. Although this

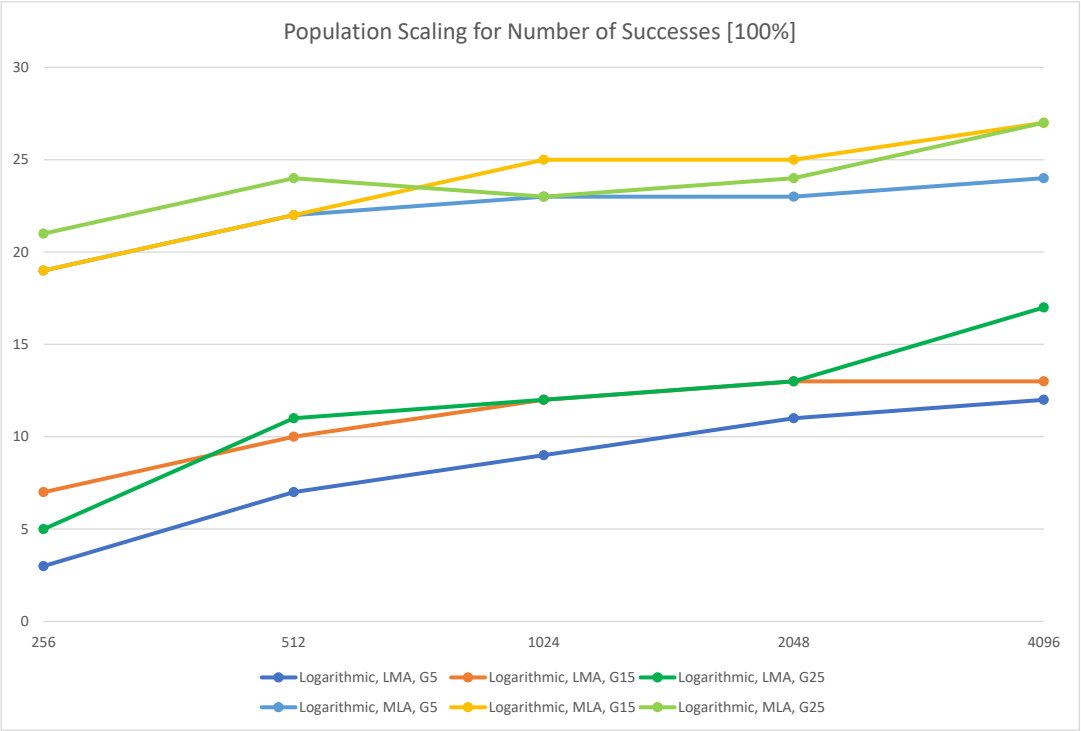

Figure 6.1: Population scaling for number of successes (100%).

confirms the general assumption that the chance of success increases with a larger population, the choice of encoding and the NRA method has a much greater impact. Considering doubling the population size roughly increases the theoretical computation cost by the same factor, the increase in success is not that impressive. Furthermore, the number of generations does not seem to further improve the success rate after 15 generations.

Running Times

The complete running times of the PE approach are given in tables B.1, B.2, B.3 and B.2.

They are color-coded based on their distance to the predefined limit of five minutes (i.e. $300s = 300000ms$). This means that a test run taking zero seconds would be green, one taking two and a half minutes would be yellow and everything at or above five minutes would be red. As can be seen in the aforementioned tables, even the biggest problems with a lot of parameters and data points, took a lot less than 30 seconds to solve. This is a very impressive result, considering how many calculations have to be performed for each chromosomes in the PE approach. This easily achieves objective C. and shows that although the PE approach is computationally

a lot more expensive than traditional approaches, it is fast enough for practical use.

However, because high-end consumer-grade hardware might not always be available, it is also important to examine the performance on more moderate hardware. Table B.5 shows the mean running times of the performance evaluation. As was stated in the setup section, it consisted of two select test problems that were evaluated on different AMP accelerators and on the CPU. The runs were repeated 10 times for a set of population sizes and for both single and double floating-point precision on the best performing configuration of the success evaluation.

Although the color-coding clearly shows a difference between the individual accelerators and the CPU, even the slower solutions only exceed the five minute limit with very large population sizes. In particular, with the exception of the population size of 8192, only the CPU run on the difficult problem using double precision took slightly more time.

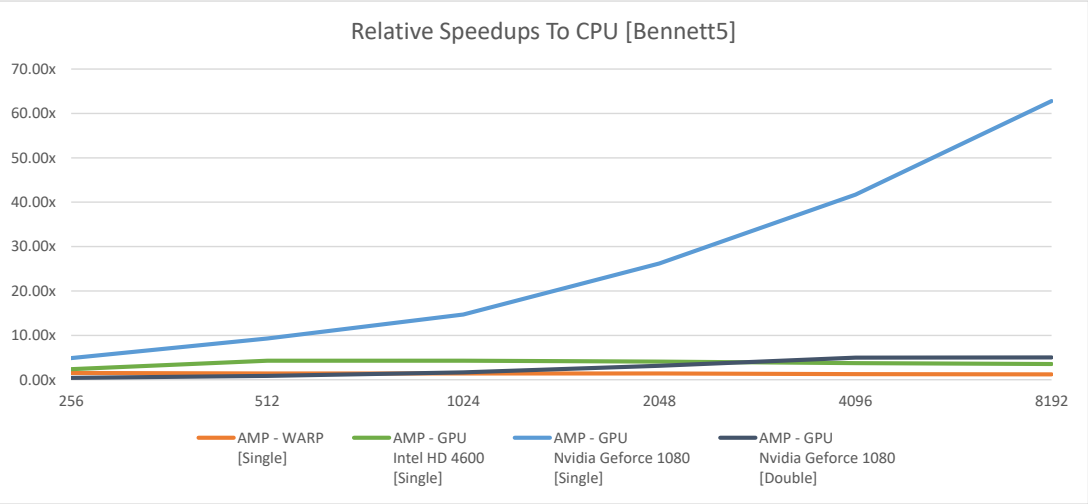

Figure 6.2: Relative speedups of an AMP accelerator to the CPU.

To make the performance differences between the hardware more apparent, table B.6 contains the speedups of each accelerator relative to the CPU. The corresponding results for the Bennett5 problem are visualized in chart 6.2. It is obvious that the difference between the Geforce GPU and the CPU is pretty significant, but also every other accelerator outperforms the CPU. This highlights an important caveat to these results. The only reason the WARP accelerator outperforms the CPU, is that it uses vector operations to speedup the execution. While it is commendable that C++ AMP can fallback to a CPU-only solution and produce satisfying results, it also shows that the CPU code is not as optimized as it could be. Due to the overhead of the DirectX emulation, a vectorized CPU solver should at least perform on-par with the WARP accelerator and probably be even faster. This would not even need to be implemented by hand. For example, libraries, such as

Intel's *Math Kernel Library (MKL)*, offer various algorithms for SVD decomposition with highly optimized vector instructions. However, even when being overly generous and granting that the CPU performance could tripled, there would still be significant advantages in using a GPU at larger population sizes.

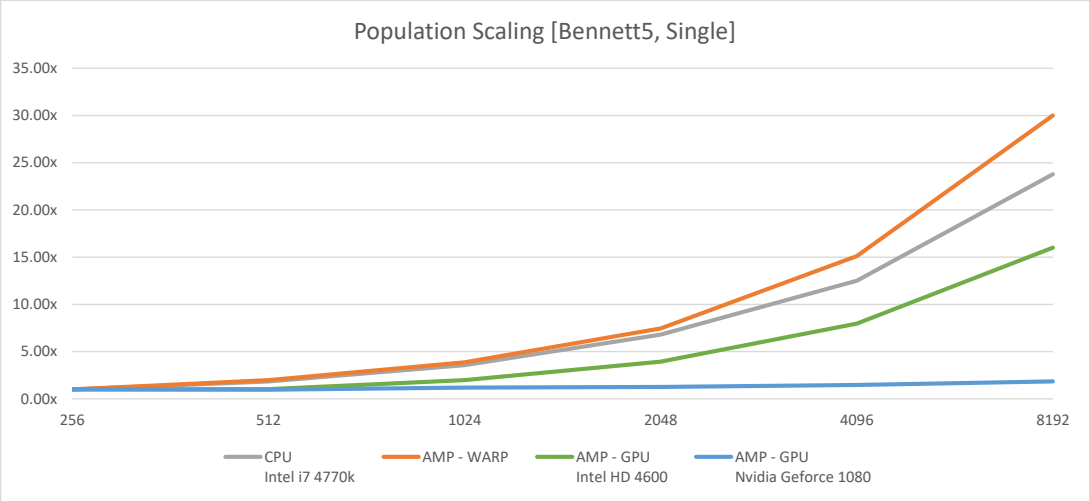

Figure 6.3: Population scaling for the increase in computation time.

The reason for this is shown in table B.7, which contains the population scaling of all accelerators. The corresponding data for the Bennett5 problem is visualized in chart 6.3 and shows how much the computation time increases when the population size is doubled. Even though all accelerators scale below the proportional factor, all but the Geforce GPU require $1.5 \times -2.0\times$ the computation time for every increase. The GTX 1080 on the other hand even with a population size of 8196 – an increase by a factor of 32 – only requires 1.85 the computation time of the 256 size population. This is not surprising, considering that GPU consists of 2560 physical cores, which can all run in parallel[33]. The fact that the GPU scales so well, shows that the AMP implementation does not contain any major flaws that prevent the software from achieving a high amount of occupancy on the graphics card.

The only case where the AMP solver does not significantly outperform the CPU, is the usage of double floating-point precision. This is a major flaw of consumer grade GPUs, which are generally optimized for single precision computation and often only have a fraction of the performance when using double precision. For the Geforce card, the GPU is reduced to $\frac{1}{32}$'th of its single precision performance. Modern CPUs on the other hand do only suffer a slight overhead when using double precision, because their registers and FPUs are designed to handle 64-bit values.

However, chart 6.4 shows that the population scaling is barely affected by the switching to double precision. This means that at sufficiently large population sizes, a GPU might still outperform a CPU.

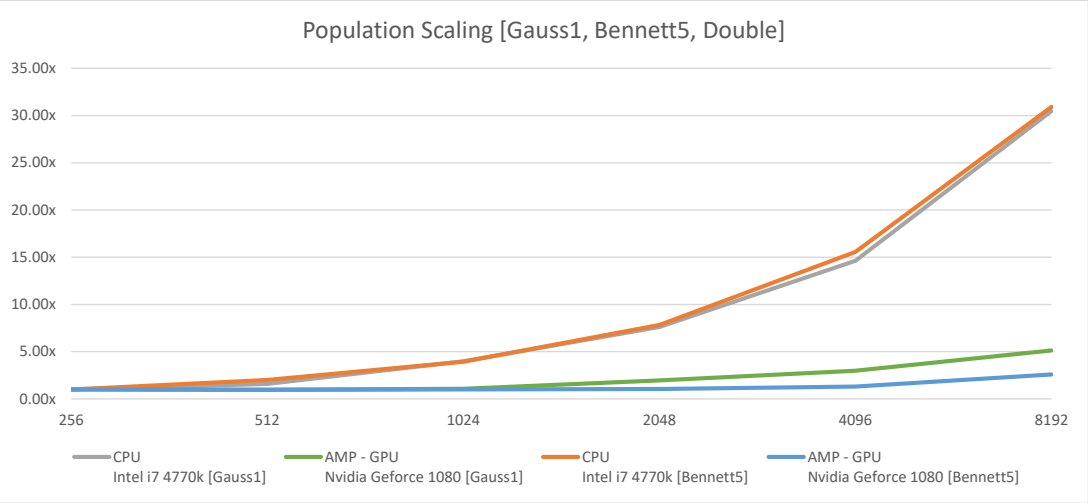

Figure 6.4: Population scaling for the increase in computation time.

Chapter 7

Conclusion

Recalling chapter 4.1.2, the goals of this thesis have been defined as follows:

I. Propose an approach to solve nonlinear regression problems independent of starting values with the following properties:

   (i) It is applicable to a wide variety of practical problems

   (ii) It is suitable for automated use and does not require any additional configuration

   (iii) It is suitable for massive parallelism with GPGPU computing and fast enough for practical use

II. Propose and examine different strategies to improve the success rate of the approach

III. Propose a possible implementation for the approach

IV. Design and perform an evaluation that can verify goals I. and II.

All of these goals were achieved.

With the Parameter Evolution approach, this thesis proposed a nontraditional genetic hybrid method that can be used to reliably solve a wide variety of nonlinear regression problems, without having to specify a set of starting values for a particular problem. It was shown how the approach can be implemented in a generic design that allows individual components to be adapted in a flexible way. It was also demonstrated how C++ AMP can be utilized as a means to achieve massive parallelism. The evaluation revealed that the PE approach is applicable to real life problems and that many of them can be solved in a matter of seconds.

The most prevalent drawback of the approach is that it gets significantly less valuable when double precision is required. Although most of the test problems could still be solved below the time limit of five minutes, double precision performance

is significantly lacking on most consumer-grade GPUs, which mitigates many disadvantages of using heterogeneous programming in the first place. However, if the single precision performance is not sufficient, there are professional GPUs which excel in double precision computing. Although they are a lot more expensive than consumer cards, they might still be more cost-effective than alternative solutions – e.g. computation clusters.

The reader may have noticed that the term "nontraditional" was used several times in this thesis. This is because the term has some significance in the broader context of this work. The topic of nonlinear regression analysis was traditionally mostly approached from a strictly mathematical angle. Only in the last several years has the field of statistics been explored by non-mathematical approaches, most notably machine learning methods.

This thesis contributes to this development by providing an alternative to strictly mathematical methods. An important distinction is the focus on computation. While a pure mathematical analysis of an approach might provide valuable insights into the raw mechanics of a method, it usually does not consider its practical implementation. Although the PE approach literally computes tens of thousands of times the work of a traditional NRA method, it can still be valuable in practice when the computation can be sufficiently parallelized. As the PE approach showed, the technologies for heterogeneous programming and massive parallelism provide opportunities to approach problems in ways that might have been considered wasteful in the past. This might lead to surprising results. In the case of the PE approach, it was shown that at least a certain amount of nonlinear regression problems can be solved by using a single configuration without having to determine any problem-specific options. This makes the approach suitable for automatic queuing of several regression models, which is something that is not possible with most conventional approaches.

Chapter 8

Outlook

There are several ways the PE approach can proceed from here. First and foremost, it was not compared to other (automatic) methods that attempt to solve the starting value problem. Most notably, the statistical programming language R – which is growing to become the new reference for statistical software – provides an entire toolbox for nonlinear regression analysis, including a variety of so called *self-starter models*[34][35].

However, there is a reason these methods were not used as a point of comparison during the evaluation of the PE approach. Although many of them are referred to as self-starting or automatic methods, most of them still require the user to specify some kind of configuration to start. Because many of them only work on specific models, it would have required to set several configurations for multiple different methods to make a test suite comparable with the PE approach. Because this would have meant including problem-specific information, which would have been subjectively determined by an experimenter, it would have defeated the purpose of comparing it to a fully automatic approach in the first place. Although a comparison could still provide some valuable insight, it would require an a elaborate evaluation, to ensure that a comparison of the approaches is actually reasonable.

Another important thing to note is that the evaluation did not really reveal the limits of the approach. By using double floating-point precision, the implementation was able to solve all 30 reference problems of the test suite in every iteration. The main reason for this is the lack of difficult reference problems for nonlinear regression analysis. Because many conventional methods rely on starting values, they are often evaluated by just making the starting values more difficult. Aside from the ones used in the evaluation, there are not many NRA reference sets available. This makes it difficult to estimate how the PE approach would perform on larger and more difficult practical problems. One option would be to seek the assistance of an experienced, professional statistician, to compare the PE approach to conventional analysis in challenging, real life scenarios.

Finally, the PE approach itself provides a lot of potential for future improvement.

In principle, every generic part of the approach could be changed to something else. There are some promising methods that could be examined in the approach. For example, Transtrum et al. promise a NRA method with higher efficiency and accuracy than the Levenberg-Marquardt algorithm[36], while Drmac et al. proposed a SVD algorithm that under certain circumstances is faster and more accurate than the Jacobi Rotation SVD algorithm[37]. Another possibility would be to move the implementation of the entire algorithm to C++ AMP, including the genetic process, to further mitigate any GPU overhead.

Appendix A

Evaluation Results

Success Rates [Discretized, LMA, Single Precision]																
Population Size	256			512			1024			2048			4096			131072
Number of Generation	5	15	25	5	15	25	5	15	25	5	15	25	5	15	25	0
L_N1_M2_DanWood	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
L_N1_M2_Misra1a	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
L_N1_M2_Misra1b	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
L_N1_M3_Chwirut1	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
L_N1_M3_Chwirut2	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
L_N1_M6_Lanczos3	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
L_N1_M8_Gauss1	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
L_N1_M8_Gauss2	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
A_N1_M2_Misra1c	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
A_N1_M2_Misra1d	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
A_N1_M4_Roszman1	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
A_N1_M5_Kirby2	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
A_N1_M5_MGH17	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
A_N1_M6_Lanczos1	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
A_N1_M6_Lanczos2	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
A_N1_M7_Hahn1	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
A_N1_M8_Gauss3	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
A_N1_M9_ENSO	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
A_N2_M3_Nelson	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
H_N1_M2_BoxBOD	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
H_N1_M3_Bennett5	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
H_N1_M3_Eckerle4	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
H_N1_M3_MGH10	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
H_N1_M3_Rat42	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
H_N1_M4_MGH09	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
H_N1_M4_Rat43	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
H_N1_M7_Thurber	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
H_N2_M3_MeyerRoth4	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
H_N2_M3_MeyerRoth5	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
H_N1_M4_MilitkyMeloun16	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
Number of Successes (> 50%)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Number of Successes (100%)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table A.1: Success rates of the PE approach for a series of test problems. This test case used the discretized encoding and the Levenberg-Marquardt Algorithm. Each test was repeated 10 times for different population sizes and generation limits. The values are color-coded percentiles for the number of successes. The rightmost column contains the results of the control configuration, which is equivalent to guessing a set of starting values 131072 times.

Success Rates [Discretized, MLA, Single Precision]																
Population Size	256			512			1024			2048			4096			131072
Number of Generation	5	15	25	5	15	25	5	15	25	5	15	25	5	15	25	0
L_N1_M2_DanWood	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
L_N1_M2_Misra1a	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
L_N1_M2_Misra1b	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
L_N1_M3_Chwirut1	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
L_N1_M3_Chwirut2	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
L_N1_M6_Lanczos3	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
L_N1_M8_Gauss1	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
L_N1_M8_Gauss2	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
A_N1_M2_Misra1c	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
A_N1_M2_Misra1d	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
A_N1_M4_Roszman1	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
A_N1_M5_Kirby2	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
A_N1_M5_MGH17	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
A_N1_M6_Lanczos1	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
A_N1_M6_Lanczos2	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
A_N1_M7_Hahn1	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
A_N1_M8_Gauss3	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
A_N1_M9_ENSO	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
A_N2_M3_Nelson	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
H_N1_M2_BoxBOD	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
H_N1_M3_Bennett5	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
H_N1_M3_Eckerle4	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
H_N1_M3_MGH10	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
H_N1_M3_Rat42	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
H_N1_M4_MGH09	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
H_N1_M4_Rat43	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
H_N1_M7_Thurber	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
H_N2_M3_MeyerRoth4	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
H_N2_M3_MeyerRoth5	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
H_N1_M4_MilitkyMeloun16	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
Number of Successes (> 50%)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Number of Successes (100%)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table A.2: Success rates of the PE approach for a series of test problems. This test case used the discretized encoding and the Modified Levenberg Algorithm. Each test was repeated 10 times for different population sizes and generation limits. The values are color-coded percentiles for the number of successes. The rightmost column contains the results of the control configuration, which is equivalent to guessing a set of starting values 131072 times.

Success Rates [Logarithmic, LMA, Single Precision]																
Population Size	256			512			1024			2048			4096			131072
Number of Generation	5	15	25	5	15	25	5	15	25	5	15	25	5	15	25	0
L_N1_M2_DanWood	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	0.00%
L_N1_M2_Misra1a	0.00%	100.00%	80.00%	0.00%	90.00%	100.00%	90.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	0.00%
L_N1_M2_Misra1b	100.00%	100.00%	80.00%	100.00%	100.00%	100.00%	60.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	0.00%
L_N1_M3_Chwirut1	30.00%	90.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	0.00%
L_N1_M3_Chwirut2	80.00%	100.00%	50.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	0.00%
L_N1_M6_Lanczos3	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
L_N1_M8_Gauss1	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
L_N1_M8_Gauss2	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
A_N1_M2_Misra1c	90.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	0.00%
A_N1_M2_Misra1d	90.00%	90.00%	80.00%	80.00%	100.00%	90.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	0.00%
A_N1_M4_Roszman1	0.00%	40.00%	0.00%	0.00%	100.00%	40.00%	0.00%	60.00%	90.00%	0.00%	90.00%	100.00%	30.00%	100.00%	100.00%	0.00%
A_N1_M5_Kirby2	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
A_N1_M5_MGH17	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	30.00%	10.00%	0.00%	60.00%	70.00%	0.00%
A_N1_M6_Lanczos1	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
A_N1_M6_Lanczos2	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
A_N1_M7_Hahn1	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
A_N1_M8_Gauss3	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
A_N1_M9_ENSO	0.00%	50.00%	90.00%	0.00%	0.00%	30.00%	0.00%	20.00%	40.00%	0.00%	100.00%	90.00%	0.00%	90.00%	100.00%	0.00%
A_N2_M3_Nelson	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	60.00%	80.00%	0.00%	90.00%	100.00%	0.00%	90.00%	100.00%	0.00%
H_N1_M2_BoxBOD	50.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	0.00%
H_N1_M3_Bennett5	100.00%	0.00%	0.00%	0.00%	0.00%	70.00%	0.00%	20.00%	0.00%	80.00%	20.00%	0.00%	10.00%	90.00%	70.00%	0.00%
H_N1_M3_Eckerle4	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
H_N1_M3_MGH10	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
H_N1_M3_Rat42	60.00%	90.00%	90.00%	100.00%	90.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	0.00%
H_N1_M4_MGH09	70.00%	80.00%	90.00%	80.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	0.00%
H_N1_M4_Rat43	0.00%	0.00%	0.00%	0.00%	0.00%	90.00%	0.00%	0.00%	30.00%	0.00%	80.00%	60.00%	20.00%	90.00%	100.00%	0.00%
H_N1_M7_Thurber	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
H_N2_M3_MeyerRoth4	0.00%	0.00%	0.00%	0.00%	0.00%	40.00%	0.00%	0.00%	0.00%	0.00%	0.00%	90.00%	0.00%	70.00%	100.00%	0.00%
H_N2_M3_MeyerRoth5	0.00%	100.00%	50.00%	80.00%	90.00%	100.00%	100.00%	100.00%	100.00%	90.00%	100.00%	90.00%	100.00%	100.00%	100.00%	0.00%
H_N1_M4_MilitkyMeloun16	70.00%	80.00%	100.00%	90.00%	100.00%	100.00%	90.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	70.00%
Number of Successes (> 50%)	9	12	11	11	13	14	12	14	14	13	16	17	12	19	19	1
Number of Successes (100%)	3	7	5	7	10	11	9	12	12	11	13	13	12	13	17	0

Table A.3: Success rates of the PE approach for a series of test problems. This test case used the logarithmic encoding and the Levenberg-Marquardt Algorithm. Each test was repeated 10 times for different population sizes and generation limits. The values are color-coded percentiles for the number of successes. The rightmost column contains the results of the control configuration, which is equivalent to guessing a set of starting values 131072 times.

Success Rates [Logarithmic, MLA, Single Precision]																
Population Size	256			512			1024			2048			4096			131072
Number of Generation	5	15	25	5	15	25	5	15	25	5	15	25	5	15	25	0
L_N1_M2_DanWood	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	90.00%
L_N1_M2_Misra1a	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	90.00%
L_N1_M2_Misra1b	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	0.00%
L_N1_M3_Chwirut1	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	0.00%
L_N1_M3_Chwirut2	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	70.00%
L_N1_M6_Lanczos3	100.00%	100.00%	100.00%	100.00%	100.00%	90.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	0.00%
L_N1_M8_Gauss1	0.00%	80.00%	20.00%	50.00%	50.00%	0.00%	0.00%	100.00%	60.00%	100.00%	80.00%	70.00%	90.00%	100.00%	90.00%	0.00%
L_N1_M8_Gauss2	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	50.00%	60.00%	50.00%	0.00%	90.00%	100.00%	0.00%
A_N1_M2_Misra1c	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	0.00%
A_N1_M2_Misra1d	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	0.00%
A_N1_M4_Roszman1	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	0.00%
A_N1_M5_Kirby2	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	0.00%
A_N1_M5_MGH17	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	0.00%
A_N1_M6_Lanczos1	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
A_N1_M6_Lanczos2	40.00%	70.00%	90.00%	100.00%	70.00%	100.00%	100.00%	100.00%	70.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	0.00%
A_N1_M7_Hahn1	90.00%	70.00%	80.00%	60.00%	70.00%	50.00%	90.00%	100.00%	100.00%	90.00%	100.00%	100.00%	100.00%	100.00%	100.00%	0.00%
A_N1_M8_Gauss3	0.00%	0.00%	0.00%	0.00%	90.00%	100.00%	0.00%	0.00%	90.00%	0.00%	50.00%	90.00%	70.00%	100.00%	100.00%	0.00%
A_N1_M9_ENSO	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	0.00%
A_N2_M3_Nelson	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	0.00%
H_N1_M2_BoxBOD	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	0.00%
H_N1_M3_Bennett5	80.00%	0.00%	60.00%	100.00%	80.00%	100.00%	80.00%	80.00%	90.00%	90.00%	100.00%	90.00%	100.00%	100.00%	100.00%	0.00%
H_N1_M3_Eckerle4	90.00%	90.00%	100.00%	90.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	0.00%
H_N1_M3_MGH10	0.00%	100.00%	80.00%	70.00%	100.00%	100.00%	100.00%	100.00%	100.00%	90.00%	100.00%	100.00%	90.00%	100.00%	100.00%	0.00%
H_N1_M3_Rat42	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	0.00%
H_N1_M4_MGH09	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	70.00%
H_N1_M4_Rat43	100.00%	80.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	0.00%
H_N1_M7_Thurber	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	50.00%	70.00%	0.00%
H_N2_M3_MeyerRoth4	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	60.00%
H_N2_M3_MeyerRoth5	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	80.00%
H_N1_M4_MilitkyMeloun16	90.00%	90.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	0.00%
Number of Successes (> 50%)	23	25	25	25	26	25	25	26	27	26	27	27	27	28	29	6
Number of Successes (100%)	19	19	21	22	22	24	23	25	23	23	25	24	24	27	27	0

Table A.4: Success rates of the PE approach for a series of test problems. This test case used the logarithmic encoding and the Modified Levenberg Algorithm. Each test was repeated 10 times for different population sizes and generation limits. The values are color-coded percentiles for the number of successes. The rightmost column contains the results of the control configuration, which is equivalent to guessing a set of starting values 131072 times.

Appendix B

Performance Analysis Results

Mean Running Times [Discretized, LMA, Single Precision]																
Population Size	256			512			1024			2048			4096			131072
Number of Generation	5	15	25	5	15	25	5	15	25	5	15	25	5	15	25	0
L_N1_M2_DanWood	35 ms	64 ms	111 ms	26 ms	72 ms	117 ms	30 ms	83 ms	135 ms	37 ms	101 ms	165 ms	50 ms	141 ms	234 ms	57 ms
L_N1_M2_Misra1a	30 ms	70 ms	120 ms	29 ms	77 ms	145 ms	32 ms	86 ms	145 ms	39 ms	106 ms	184 ms	51 ms	149 ms	282 ms	53 ms
L_N1_M2_Misra1b	29 ms	68 ms	122 ms	29 ms	78 ms	139 ms	32 ms	88 ms	149 ms	39 ms	113 ms	199 ms	52 ms	165 ms	325 ms	71 ms
L_N1_M3_Chwirut1	150 ms	377 ms	685 ms	152 ms	402 ms	752 ms	166 ms	448 ms	727 ms	174 ms	471 ms	769 ms	191 ms	526 ms	890 ms	386 ms
L_N1_M3_Chwirut2	70 ms	173 ms	306 ms	68 ms	192 ms	374 ms	73 ms	197 ms	325 ms	81 ms	222 ms	364 ms	98 ms	269 ms	430 ms	161 ms
L_N1_M6_Lanczos3	237 ms	601 ms	1059 ms	232 ms	626 ms	1097 ms	250 ms	669 ms	1086 ms	252 ms	670 ms	1094 ms	493 ms	1319 ms	2153 ms	1612 ms
L_N1_M8_Gauss1	735 ms	1914 ms	3430 ms	750 ms	1878 ms	3384 ms	756 ms	2006 ms	3308 ms	1468 ms	3982 ms	6402 ms	2243 ms	5935 ms	9778 ms	8261 ms
L_N1_M8_Gauss2	744 ms	1961 ms	3320 ms	756 ms	1962 ms	3392 ms	755 ms	2022 ms	3253 ms	1464 ms	3962 ms	6469 ms	2238 ms	5977 ms	10408 ms	8240 ms
A_N1_M2_Misra1c	27 ms	72 ms	119 ms	33 ms	77 ms	126 ms	32 ms	86 ms	138 ms	42 ms	105 ms	162 ms	51 ms	146 ms	248 ms	53 ms
A_N1_M2_Misra1d	28 ms	73 ms	126 ms	31 ms	76 ms	130 ms	31 ms	86 ms	139 ms	43 ms	105 ms	169 ms	51 ms	145 ms	272 ms	62 ms
A_N1_M4_Roszman1	95 ms	236 ms	418 ms	94 ms	241 ms	397 ms	95 ms	253 ms	411 ms	105 ms	270 ms	427 ms	123 ms	339 ms	552 ms	301 ms
A_N1_M5_Kirby2	232 ms	621 ms	1077 ms	248 ms	637 ms	1065 ms	249 ms	667 ms	1058 ms	258 ms	682 ms	1068 ms	288 ms	770 ms	1269 ms	1297 ms
A_N1_M5_MGH17	154 ms	411 ms	715 ms	157 ms	417 ms	785 ms	161 ms	433 ms	682 ms	170 ms	466 ms	706 ms	192 ms	520 ms	845 ms	785 ms
A_N1_M6_Lanczos1	250 ms	626 ms	1119 ms	238 ms	635 ms	1122 ms	247 ms	669 ms	1054 ms	250 ms	695 ms	1060 ms	492 ms	1311 ms	2139 ms	1607 ms
A_N1_M6_Lanczos2	253 ms	626 ms	1116 ms	242 ms	635 ms	1151 ms	254 ms	669 ms	1084 ms	252 ms	693 ms	1093 ms	501 ms	1464 ms	2137 ms	1619 ms
A_N1_M7_Hahn1	580 ms	1449 ms	2380 ms	549 ms	1467 ms	2491 ms	570 ms	1507 ms	2404 ms	1080 ms	2901 ms	4721 ms	1682 ms	4680 ms	7201 ms	6254 ms
A_N1_M8_Gauss3	737 ms	1961 ms	3262 ms	756 ms	1969 ms	3448 ms	749 ms	2007 ms	3254 ms	1457 ms	3908 ms	6390 ms	2215 ms	6395 ms	9676 ms	8266 ms
A_N1_M9_ENSO	925 ms	2343 ms	3771 ms	902 ms	2351 ms	4233 ms	904 ms	2411 ms	3962 ms	1756 ms	4686 ms	7747 ms	2651 ms	7465 ms	11610 ms	10047 ms
A_N2_M3_Nelson	106 ms	267 ms	429 ms	102 ms	272 ms	485 ms	109 ms	298 ms	484 ms	120 ms	324 ms	501 ms	127 ms	391 ms	569 ms	255 ms
H_N1_M2_BoxBOD	30 ms	63 ms	100 ms	26 ms	68 ms	120 ms	28 ms	78 ms	140 ms	35 ms	95 ms	149 ms	45 ms	143 ms	208 ms	53 ms
H_N1_M3_Bennett5	126 ms	323 ms	511 ms	123 ms	329 ms	546 ms	133 ms	357 ms	586 ms	141 ms	378 ms	625 ms	149 ms	443 ms	654 ms	302 ms
H_N1_M3_Eckerle4	60 ms	158 ms	250 ms	63 ms	164 ms	259 ms	63 ms	171 ms	282 ms	70 ms	190 ms	314 ms	83 ms	270 ms	372 ms	128 ms
H_N1_M3_MGH10	53 ms	134 ms	209 ms	51 ms	135 ms	219 ms	54 ms	147 ms	240 ms	61 ms	164 ms	269 ms	71 ms	239 ms	324 ms	118 ms
H_N1_M3_Rat42	47 ms	123 ms	195 ms	50 ms	127 ms	214 ms	51 ms	134 ms	224 ms	57 ms	155 ms	254 ms	67 ms	216 ms	321 ms	99 ms
H_N1_M4_MGH09	85 ms	217 ms	352 ms	85 ms	221 ms	362 ms	86 ms	232 ms	381 ms	93 ms	249 ms	408 ms	104 ms	331 ms	467 ms	267 ms
H_N1_M4_Rat43	88 ms	227 ms	372 ms	88 ms	231 ms	397 ms	90 ms	242 ms	400 ms	96 ms	259 ms	425 ms	109 ms	358 ms	483 ms	280 ms
H_N1_M7_Thurber	363 ms	970 ms	1688 ms	372 ms	977 ms	1591 ms	377 ms	1000 ms	1636 ms	726 ms	1930 ms	3152 ms	1143 ms	3345 ms	4967 ms	4062 ms
H_N2_M3_MeyerRoth4	54 ms	141 ms	256 ms	54 ms	145 ms	237 ms	58 ms	155 ms	254 ms	64 ms	173 ms	284 ms	79 ms	265 ms	338 ms	127 ms
H_N2_M3_MeyerRoth5	54 ms	141 ms	252 ms	56 ms	147 ms	237 ms	57 ms	155 ms	255 ms	64 ms	174 ms	285 ms	78 ms	273 ms	339 ms	128 ms
H_N1_M4_MilitkyMeloun16	90 ms	222 ms	368 ms	86 ms	229 ms	373 ms	89 ms	237 ms	393 ms	94 ms	257 ms	429 ms	108 ms	319 ms	478 ms	268 ms

Table B.1: Mean running times of the PE approach for a series of test problems. This test case used the discretized encoding and the Levenberg-Marquardt Algorithm. Each test was repeated 10 times for different population sizes and generation limits. The values are the mean running times in milliseconds. They are color-coded from green (0 ms) to red (300000 ms). The rightmost column contains the results of the control configuration, which is equivalent to guessing a set of starting values 131072 times.

Mean Running Times [Discretized, MLA, Single Precision]																
Population Size	256			512			1024			2048			4096			131072
Number of Generation	5	15	25	5	15	25	5	15	25	5	15	25	5	15	25	0
L_N1_M2_DanWood	24 ms	62 ms	132 ms	25 ms	67 ms	117 ms	29 ms	93 ms	140 ms	38 ms	94 ms	164 ms	51 ms	142 ms	214 ms	68 ms
L_N1_M2_Misra1a	27 ms	71 ms	133 ms	27 ms	75 ms	133 ms	31 ms	99 ms	146 ms	38 ms	102 ms	179 ms	50 ms	155 ms	237 ms	62 ms
L_N1_M2_Misra1b	27 ms	69 ms	132 ms	28 ms	74 ms	139 ms	32 ms	99 ms	154 ms	37 ms	101 ms	183 ms	49 ms	173 ms	242 ms	63 ms
L_N1_M3_Chwirut1	203 ms	549 ms	935 ms	203 ms	542 ms	934 ms	259 ms	742 ms	1194 ms	278 ms	733 ms	1286 ms	302 ms	858 ms	1306 ms	739 ms
L_N1_M3_Chwirut2	81 ms	214 ms	365 ms	80 ms	213 ms	369 ms	85 ms	234 ms	406 ms	91 ms	243 ms	399 ms	120 ms	348 ms	532 ms	245 ms
L_N1_M6_Lanczos3	254 ms	700 ms	1191 ms	253 ms	678 ms	1171 ms	279 ms	727 ms	1150 ms	269 ms	719 ms	1227 ms	533 ms	1491 ms	2347 ms	1806 ms
L_N1_M8_Gauss1	1721 ms	4620 ms	7649 ms	1825 ms	4873 ms	8114 ms	1911 ms	5131 ms	8252 ms	3362 ms	8720 ms	14637 ms	5624 ms	15113 ms	24491 ms	21599 ms
L_N1_M8_Gauss2	1720 ms	4631 ms	7658 ms	1849 ms	4873 ms	8104 ms	1925 ms	5030 ms	8259 ms	3383 ms	9034 ms	14398 ms	5640 ms	14953 ms	24503 ms	21627 ms
A_N1_M2_Misra1c	30 ms	72 ms	127 ms	34 ms	78 ms	132 ms	35 ms	88 ms	150 ms	41 ms	110 ms	168 ms	53 ms	150 ms	234 ms	71 ms
A_N1_M2_Misra1d	26 ms	68 ms	120 ms	31 ms	72 ms	129 ms	33 ms	87 ms	148 ms	38 ms	107 ms	161 ms	49 ms	154 ms	266 ms	63 ms
A_N1_M4_Roszman1	95 ms	253 ms	447 ms	110 ms	257 ms	449 ms	102 ms	269 ms	466 ms	108 ms	306 ms	463 ms	131 ms	390 ms	646 ms	400 ms
A_N1_M5_Kirby2	376 ms	1035 ms	1738 ms	449 ms	1016 ms	1736 ms	487 ms	1305 ms	2193 ms	506 ms	1405 ms	2155 ms	551 ms	1551 ms	2525 ms	2687 ms
A_N1_M5_MGH17	171 ms	456 ms	797 ms	211 ms	459 ms	800 ms	180 ms	481 ms	822 ms	185 ms	523 ms	794 ms	229 ms	659 ms	1053 ms	1010 ms
A_N1_M6_Lanczos1	250 ms	678 ms	1163 ms	306 ms	673 ms	1167 ms	261 ms	709 ms	1187 ms	269 ms	753 ms	1149 ms	531 ms	1483 ms	2483 ms	1816 ms
A_N1_M6_Lanczos2	251 ms	669 ms	1157 ms	298 ms	673 ms	1167 ms	261 ms	690 ms	1189 ms	268 ms	752 ms	1152 ms	531 ms	1482 ms	2567 ms	1821 ms
A_N1_M7_Hahn1	1045 ms	2777 ms	4640 ms	1419 ms	3569 ms	5915 ms	1408 ms	3697 ms	6046 ms	2453 ms	6561 ms	10299 ms	3933 ms	10515 ms	17326 ms	15780 ms
A_N1_M8_Gauss3	1717 ms	4724 ms	7720 ms	2128 ms	4868 ms	8087 ms	1921 ms	5082 ms	8263 ms	3367 ms	9071 ms	13939 ms	5648 ms	15108 ms	24494 ms	21573 ms
A_N1_M9_ENSO	1693 ms	4516 ms	7718 ms	2419 ms	5544 ms	9228 ms	2223 ms	5794 ms	9645 ms	3919 ms	10412 ms	16301 ms	6124 ms	16779 ms	27077 ms	24981 ms
A_N2_M3_Nelson	133 ms	351 ms	611 ms	166 ms	362 ms	620 ms	152 ms	400 ms	698 ms	178 ms	500 ms	762 ms	199 ms	570 ms	934 ms	498 ms
H_N1_M2_BoxBOD	22 ms	59 ms	107 ms	32 ms	63 ms	113 ms	27 ms	80 ms	130 ms	32 ms	97 ms	146 ms	46 ms	138 ms	233 ms	48 ms
H_N1_M3_Bennett5	157 ms	417 ms	724 ms	199 ms	422 ms	735 ms	194 ms	513 ms	888 ms	211 ms	594 ms	905 ms	232 ms	668 ms	1113 ms	615 ms
H_N1_M3_Eckerle4	64 ms	172 ms	299 ms	86 ms	176 ms	306 ms	70 ms	187 ms	327 ms	76 ms	217 ms	331 ms	98 ms	284 ms	487 ms	179 ms
H_N1_M3_MGH10	49 ms	131 ms	229 ms	67 ms	136 ms	237 ms	54 ms	146 ms	262 ms	60 ms	177 ms	266 ms	74 ms	217 ms	364 ms	133 ms
H_N1_M3_Rat42	44 ms	118 ms	207 ms	59 ms	121 ms	215 ms	49 ms	131 ms	230 ms	55 ms	161 ms	242 ms	68 ms	203 ms	366 ms	95 ms
H_N1_M4_MGH09	81 ms	214 ms	371 ms	102 ms	218 ms	379 ms	89 ms	229 ms	402 ms	97 ms	261 ms	398 ms	113 ms	314 ms	526 ms	272 ms
H_N1_M4_Rat43	86 ms	230 ms	401 ms	114 ms	234 ms	408 ms	92 ms	249 ms	437 ms	100 ms	278 ms	426 ms	113 ms	330 ms	560 ms	310 ms
H_N1_M7_Thurber	409 ms	1092 ms	1880 ms	507 ms	1096 ms	1888 ms	426 ms	1128 ms	1958 ms	869 ms	2319 ms	3630 ms	1364 ms	3573 ms	5909 ms	4826 ms
H_N2_M3_MeyerRoth4	55 ms	145 ms	255 ms	70 ms	167 ms	283 ms	62 ms	161 ms	279 ms	65 ms	191 ms	287 ms	78 ms	221 ms	390 ms	151 ms
H_N2_M3_MeyerRoth5	54 ms	145 ms	255 ms	69 ms	150 ms	283 ms	59 ms	163 ms	279 ms	66 ms	191 ms	287 ms	78 ms	245 ms	380 ms	149 ms
H_N1_M4_MilitkyMeloun16	86 ms	230 ms	400 ms	92 ms	263 ms	429 ms	95 ms	247 ms	427 ms	98 ms	281 ms	426 ms	117 ms	342 ms	542 ms	312 ms

Table B.2: Mean running times of the PE approach for a series of test problems. This test case used the discretized encoding and the Modified Levenberg Algorithm. Each test was repeated 10 times for different population sizes and generation limits. The values are the mean running times in milliseconds. They are color-coded from green (0 ms) to red (300000 ms). The rightmost column contains the results of the control configuration, which is equivalent to guessing a set of starting values 131072 times.

Mean Running Times [Logarithmic, LMA, Single Precision]																
Population Size	256			512			1024			2048			4096			131072
Number of Generation	5	15	25	5	15	25	5	15	25	5	15	25	5	15	25	0
L_N1_M2_DanWood	26 ms	64 ms	103 ms	25 ms	68 ms	110 ms	28 ms	76 ms	125 ms	35 ms	95 ms	154 ms	47 ms	132 ms	215 ms	72 ms
L_N1_M2_Misra1a	29 ms	69 ms	111 ms	27 ms	72 ms	119 ms	30 ms	83 ms	135 ms	36 ms	99 ms	164 ms	50 ms	136 ms	239 ms	57 ms
L_N1_M2_Misra1b	29 ms	69 ms	112 ms	28 ms	73 ms	120 ms	32 ms	81 ms	135 ms	37 ms	102 ms	164 ms	49 ms	146 ms	243 ms	67 ms
L_N1_M3_Chwirut1	147 ms	379 ms	614 ms	143 ms	381 ms	622 ms	158 ms	421 ms	685 ms	166 ms	441 ms	719 ms	181 ms	492 ms	804 ms	405 ms
L_N1_M3_Chwirut2	65 ms	175 ms	283 ms	67 ms	178 ms	290 ms	70 ms	188 ms	307 ms	79 ms	215 ms	345 ms	95 ms	258 ms	419 ms	163 ms
L_N1_M6_Lanczos3	232 ms	603 ms	979 ms	228 ms	605 ms	985 ms	237 ms	634 ms	1030 ms	240 ms	642 ms	1042 ms	473 ms	1260 ms	2052 ms	1571 ms
L_N1_M8_Gauss1	716 ms	1883 ms	3052 ms	705 ms	1877 ms	3061 ms	720 ms	1918 ms	3116 ms	1462 ms	3936 ms	6408 ms	2223 ms	5961 ms	9719 ms	8223 ms
L_N1_M8_Gauss2	708 ms	1870 ms	3049 ms	704 ms	1882 ms	3060 ms	720 ms	1918 ms	3114 ms	1461 ms	3939 ms	6405 ms	2230 ms	5960 ms	9711 ms	8211 ms
A_N1_M2_Misra1c	28 ms	69 ms	111 ms	27 ms	72 ms	118 ms	30 ms	82 ms	133 ms	36 ms	99 ms	162 ms	49 ms	135 ms	221 ms	57 ms
A_N1_M2_Misra1d	28 ms	68 ms	111 ms	27 ms	72 ms	118 ms	30 ms	83 ms	133 ms	36 ms	99 ms	162 ms	49 ms	135 ms	231 ms	62 ms
A_N1_M4_Roszman1	91 ms	225 ms	367 ms	86 ms	230 ms	373 ms	90 ms	242 ms	391 ms	96 ms	257 ms	421 ms	116 ms	316 ms	516 ms	306 ms
A_N1_M5_Kirby2	234 ms	592 ms	964 ms	226 ms	603 ms	976 ms	237 ms	635 ms	1025 ms	244 ms	654 ms	1064 ms	273 ms	730 ms	1191 ms	1304 ms
A_N1_M5_MGH17	157 ms	393 ms	638 ms	148 ms	396 ms	643 ms	153 ms	412 ms	666 ms	160 ms	430 ms	710 ms	182 ms	492 ms	805 ms	795 ms
A_N1_M6_Lanczos1	225 ms	599 ms	973 ms	226 ms	603 ms	980 ms	237 ms	634 ms	1024 ms	238 ms	638 ms	1038 ms	470 ms	1252 ms	2040 ms	1596 ms
A_N1_M6_Lanczos2	225 ms	599 ms	973 ms	226 ms	604 ms	981 ms	237 ms	633 ms	1024 ms	238 ms	638 ms	1038 ms	469 ms	1251 ms	2042 ms	1588 ms
A_N1_M7_Hahn1	533 ms	1396 ms	2270 ms	530 ms	1415 ms	2295 ms	547 ms	1459 ms	2355 ms	1099 ms	3009 ms	4840 ms	1706 ms	4574 ms	7438 ms	6263 ms
A_N1_M8_Gauss3	702 ms	1874 ms	3051 ms	703 ms	1888 ms	3059 ms	719 ms	1917 ms	3114 ms	1457 ms	3944 ms	6388 ms	2229 ms	5945 ms	9717 ms	8224 ms
A_N1_M9_ENSO	875 ms	2292 ms	3755 ms	880 ms	2349 ms	3867 ms	904 ms	2407 ms	3917 ms	1755 ms	4709 ms	7680 ms	2667 ms	7111 ms	11566 ms	9951 ms
A_N2_M3_Nelson	99 ms	260 ms	415 ms	99 ms	260 ms	424 ms	107 ms	281 ms	459 ms	116 ms	307 ms	500 ms	129 ms	351 ms	571 ms	249 ms
H_N1_M2_BoxBOD	22 ms	61 ms	98 ms	24 ms	63 ms	105 ms	27 ms	73 ms	120 ms	32 ms	91 ms	148 ms	45 ms	125 ms	207 ms	50 ms
H_N1_M3_Bennett5	118 ms	308 ms	496 ms	116 ms	309 ms	504 ms	127 ms	339 ms	552 ms	134 ms	359 ms	585 ms	150 ms	403 ms	657 ms	329 ms
H_N1_M3_Eckerle4	61 ms	151 ms	245 ms	58 ms	155 ms	252 ms	62 ms	164 ms	268 ms	68 ms	186 ms	297 ms	83 ms	228 ms	372 ms	133 ms
H_N1_M3_MGH10	50 ms	127 ms	205 ms	48 ms	130 ms	211 ms	52 ms	140 ms	228 ms	57 ms	158 ms	257 ms	72 ms	196 ms	320 ms	117 ms
H_N1_M3_Rat42	46 ms	118 ms	189 ms	45 ms	120 ms	196 ms	48 ms	131 ms	212 ms	54 ms	148 ms	242 ms	67 ms	187 ms	304 ms	99 ms
H_N1_M4_MGH09	83 ms	208 ms	335 ms	78 ms	210 ms	342 ms	83 ms	221 ms	361 ms	88 ms	238 ms	387 ms	105 ms	285 ms	463 ms	268 ms
H_N1_M4_Rat43	87 ms	218 ms	352 ms	83 ms	222 ms	358 ms	86 ms	230 ms	378 ms	92 ms	248 ms	405 ms	109 ms	298 ms	485 ms	274 ms
H_N1_M7_Thurber	354 ms	933 ms	1506 ms	352 ms	934 ms	1519 ms	361 ms	961 ms	1561 ms	695 ms	1846 ms	3010 ms	1175 ms	3153 ms	5107 ms	4058 ms
H_N2_M3_MeyerRoth4	53 ms	135 ms	219 ms	52 ms	139 ms	225 ms	55 ms	147 ms	240 ms	61 ms	166 ms	270 ms	76 ms	208 ms	344 ms	120 ms
H_N2_M3_MeyerRoth5	50 ms	135 ms	219 ms	52 ms	139 ms	225 ms	55 ms	147 ms	240 ms	61 ms	165 ms	271 ms	75 ms	207 ms	340 ms	119 ms
H_N1_M4_MilitkyMeloun16	85 ms	214 ms	347 ms	82 ms	218 ms	353 ms	85 ms	229 ms	372 ms	92 ms	245 ms	399 ms	107 ms	293 ms	478 ms	268 ms

Table B.3: Mean running times of the PE approach for a series of test problems. This test case used the logarithmic encoding and the Levenberg-Marquardt Algorithm. Each test was repeated 10 times for different population sizes and generation limits. The values are the mean running times in milliseconds. They are color-coded from green (0 ms) to red (300000 ms). The rightmost column contains the results of the control configuration, which is equivalent to guessing a set of starting values 131072 times.

Mean Running Times [Logarithmic, MLA, Single Precision]																
Population Size	256			512			1024			2048			4096			131072
Number of Generation	5	15	25	5	15	25	5	15	25	5	15	25	5	15	25	0
L_N1_M2_DanWood	35 ms	62 ms	99 ms	25 ms	66 ms	106 ms	28 ms	75 ms	122 ms	33 ms	93 ms	150 ms	47 ms	129 ms	209 ms	64 ms
L_N1_M2_Misra1a	30 ms	70 ms	114 ms	29 ms	75 ms	122 ms	32 ms	84 ms	136 ms	37 ms	101 ms	163 ms	51 ms	139 ms	236 ms	71 ms
L_N1_M2_Misra1b	28 ms	70 ms	113 ms	28 ms	74 ms	120 ms	31 ms	83 ms	135 ms	36 ms	100 ms	163 ms	50 ms	144 ms	239 ms	63 ms
L_N1_M3_Chwirut1	205 ms	534 ms	869 ms	201 ms	540 ms	877 ms	254 ms	681 ms	1100 ms	273 ms	730 ms	1182 ms	298 ms	800 ms	1304 ms	720 ms
L_N1_M3_Chwirut2	78 ms	208 ms	338 ms	78 ms	213 ms	347 ms	85 ms	224 ms	363 ms	92 ms	243 ms	398 ms	119 ms	321 ms	525 ms	245 ms
L_N1_M6_Lanczos3	256 ms	671 ms	1091 ms	251 ms	675 ms	1101 ms	260 ms	694 ms	1121 ms	265 ms	710 ms	1151 ms	522 ms	1400 ms	2269 ms	1787 ms
L_N1_M8_Gauss1	1708 ms	4605 ms	7474 ms	1824 ms	4871 ms	7905 ms	1867 ms	5015 ms	8130 ms	3245 ms	8672 ms	14080 ms	5431 ms	14480 ms	23472 ms	21420 ms
L_N1_M8_Gauss2	1710 ms	4608 ms	7479 ms	1823 ms	4902 ms	7919 ms	1878 ms	5020 ms	8138 ms	3256 ms	8656 ms	14078 ms	5418 ms	14474 ms	23482 ms	21415 ms
A_N1_M2_Misra1c	34 ms	73 ms	117 ms	31 ms	77 ms	124 ms	33 ms	87 ms	140 ms	40 ms	104 ms	168 ms	53 ms	140 ms	224 ms	70 ms
A_N1_M2_Misra1d	29 ms	68 ms	110 ms	26 ms	72 ms	117 ms	30 ms	81 ms	132 ms	36 ms	98 ms	160 ms	48 ms	135 ms	227 ms	63 ms
A_N1_M4_Roszman1	100 ms	252 ms	411 ms	96 ms	258 ms	417 ms	100 ms	269 ms	436 ms	106 ms	285 ms	463 ms	131 ms	352 ms	575 ms	409 ms
A_N1_M5_Kirby2	384 ms	1001 ms	1633 ms	378 ms	1010 ms	1637 ms	482 ms	1275 ms	2069 ms	499 ms	1325 ms	2155 ms	544 ms	1452 ms	2353 ms	2668 ms
A_N1_M5_MGH17	178 ms	454 ms	739 ms	171 ms	459 ms	745 ms	178 ms	472 ms	768 ms	182 ms	487 ms	793 ms	225 ms	605 ms	984 ms	1020 ms
A_N1_M6_Lanczos1	249 ms	667 ms	1086 ms	253 ms	679 ms	1093 ms	258 ms	684 ms	1116 ms	264 ms	704 ms	1146 ms	525 ms	1397 ms	2273 ms	1786 ms
A_N1_M6_Lanczos2	249 ms	667 ms	1086 ms	253 ms	673 ms	1094 ms	258 ms	683 ms	1116 ms	265 ms	704 ms	1147 ms	524 ms	1395 ms	2273 ms	1792 ms
A_N1_M7_Hahn1	1073 ms	2827 ms	4561 ms	1343 ms	3574 ms	5803 ms	1383 ms	3697 ms	6004 ms	2408 ms	6390 ms	10385 ms	3837 ms	10235 ms	16562 ms	15704 ms
A_N1_M8_Gauss3	1704 ms	4607 ms	7468 ms	1832 ms	4884 ms	7908 ms	1872 ms	5028 ms	8134 ms	3257 ms	8663 ms	14076 ms	5426 ms	14528 ms	23427 ms	21367 ms
A_N1_M9_ENSO	1690 ms	4509 ms	7333 ms	2079 ms	5543 ms	8996 ms	2172 ms	5818 ms	9437 ms	3767 ms	10043 ms	16317 ms	5951 ms	15917 ms	25824 ms	24818 ms
A_N2_M3_Nelson	139 ms	350 ms	569 ms	133 ms	356 ms	577 ms	154 ms	406 ms	643 ms	176 ms	468 ms	759 ms	198 ms	531 ms	864 ms	506 ms
H_N1_M2_BoxBOD	22 ms	59 ms	97 ms	23 ms	63 ms	103 ms	26 ms	72 ms	118 ms	32 ms	89 ms	146 ms	44 ms	124 ms	204 ms	52 ms
H_N1_M3_Bennett5	159 ms	420 ms	678 ms	158 ms	422 ms	685 ms	192 ms	513 ms	834 ms	208 ms	555 ms	903 ms	231 ms	621 ms	1012 ms	571 ms
H_N1_M3_Eckerle4	67 ms	172 ms	279 ms	66 ms	176 ms	286 ms	69 ms	187 ms	303 ms	75 ms	203 ms	330 ms	97 ms	265 ms	432 ms	184 ms
H_N1_M3_MGH10	52 ms	131 ms	214 ms	51 ms	136 ms	220 ms	54 ms	145 ms	237 ms	59 ms	162 ms	265 ms	73 ms	201 ms	328 ms	128 ms
H_N1_M3_Rat42	47 ms	117 ms	191 ms	45 ms	121 ms	197 ms	48 ms	131 ms	213 ms	54 ms	148 ms	242 ms	67 ms	185 ms	304 ms	99 ms
H_N1_M4_MGH09	84 ms	214 ms	348 ms	81 ms	218 ms	354 ms	85 ms	229 ms	371 ms	90 ms	245 ms	399 ms	107 ms	291 ms	473 ms	270 ms
H_N1_M4_Rat43	92 ms	237 ms	373 ms	87 ms	233 ms	380 ms	91 ms	244 ms	397 ms	96 ms	261 ms	425 ms	113 ms	305 ms	501 ms	318 ms
H_N1_M7_Thurber	413 ms	1091 ms	1775 ms	410 ms	1097 ms	1781 ms	421 ms	1130 ms	1822 ms	847 ms	2289 ms	3699 ms	1376 ms	3727 ms	6069 ms	4809 ms
H_N2_M3_MeyerRoth4	57 ms	145 ms	237 ms	56 ms	149 ms	243 ms	59 ms	163 ms	258 ms	64 ms	177 ms	287 ms	79 ms	215 ms	351 ms	148 ms
H_N2_M3_MeyerRoth5	55 ms	145 ms	236 ms	55 ms	150 ms	243 ms	59 ms	159 ms	259 ms	65 ms	177 ms	289 ms	78 ms	214 ms	353 ms	146 ms
H_N1_M4_MilitkyMeloun16	90 ms	229 ms	374 ms	87 ms	234 ms	381 ms	92 ms	245 ms	397 ms	97 ms	262 ms	429 ms	113 ms	306 ms	502 ms	320 ms

Table B.4: Mean running times of the PE approach for a series of test problems. This test case used the logarithmic encoding and the Modified Levenberg Algorithm. Each test was repeated 10 times for different population sizes and generation limits. The values are the mean running times in milliseconds. They are color-coded from green (0 ms) to red (300000 ms). The rightmost column contains the results of the control configuration, which is equivalent to guessing a set of starting values 131072 times.

Mean Running Times [Logarithmic, MLA]													
Precision		Single						Double					
Population Size / No. Generations		256 / 15	512 / 15	1024 / 15	2048 / 15	4096 / 15	8192 / 15	256 / 15	512 / 15	1024 / 15	2048 / 15	4096 / 15	8192 / 15
CPU Intel i7 4770k	L_N1_M8_Gauss1	19173 ms	28361 ms	44636 ms	123289 ms	244429 ms	694222 ms	24128 ms	38545 ms	96389 ms	183545 ms	352924 ms	735400 ms
	H_N1_M3_Bennett5	2165 ms	3995 ms	7721 ms	14716 ms	27075 ms	51487 ms	2372 ms	4730 ms	9325 ms	18566 ms	36885 ms	73334 ms
AMP - GPU Nvidia Geforce 1080	L_N1_M8_Gauss1	4653 ms	4824 ms	4971 ms	8644 ms	14469 ms	24843 ms	34854 ms	34148 ms	37564 ms	68012 ms	103764 ms	178818 ms
	H_N1_M3_Bennett5	444 ms	429 ms	526 ms	562 ms	649 ms	820 ms	5633 ms	5398 ms	5609 ms	5938 ms	7408 ms	14586 ms
AMP - GPU Intel HD 4600	L_N1_M8_Gauss1	12799 ms	14488 ms	28875 ms	58039 ms	121881 ms	241325 ms						
	H_N1_M3_Bennett5	908 ms	936 ms	1799 ms	3580 ms	7232 ms	14536 ms						
AMP - WARP	L_N1_M8_Gauss1	16869 ms	32307 ms	62337 ms	121852 ms	242662 ms	486602 ms						
	H_N1_M3_Bennett5	1415 ms	2800 ms	5472 ms	10545 ms	21400 ms	42452 ms						

Table B.5: Mean running times of the PE approach for two test problems of different size. Each test was repeated 10 times for different parallelization methods, as well as for different population sizes, generation limits and floating point precisions. The values are the mean running times in milliseconds. They are color-coded from green (0 ms) to red (300000 ms).

Relative Speedups to CPU [Logarithmic, MLA]

Precision		Single						Double					
Population Size / No. Generations		256 / 15	512 / 15	1024 / 15	2048 / 15	4096 / 15	8192 / 15	256 / 15	512 / 15	1024 / 15	2048 / 15	4096 / 15	8192 / 15
CPU Intel i7 4770k	L_N1_M8_Gauss1	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x
	H_N1_M3_Bennett5	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x
AMP - GPU Nvidia Geforce 1080	L_N1_M8_Gauss1	4.12x	5.88x	8.98x	14.26x	16.89x	27.94x	0.69x	1.13x	2.57x	2.70x	3.40x	4.11x
	H_N1_M3_Bennett5	4.88x	9.31x	14.68x	26.19x	41.72x	62.79x	0.42x	0.88x	1.66x	3.13x	4.98x	5.03x
AMP - GPU Intel HD 4600	L_N1_M8_Gauss1	1.50x	1.96x	1.55x	2.12x	2.01x	2.88x						
	H_N1_M3_Bennett5	2.38x	4.27x	4.29x	4.11x	3.74x	3.54x						
AMP - WARP	L_N1_M8_Gauss1	1.14x	0.88x	0.72x	1.01x	1.01x	1.43x						
	H_N1_M3_Bennett5	1.53x	1.43x	1.41x	1.40x	1.27x	1.21x						

Table B.6: Relative speedups of the PE approach for two test problems of different size. Each test was repeated 10 times for different parallelization methods, as well as for different population sizes and floating point precisions. The values specify the relative speedup in computation time of a parallelization method compared to the CPU. They are color-coded from red (smallest value) to green (largest value).

Population Scaling [Logarithmic, MLA]													
Precision		Single						Double					
Population Size / No. Generations		256 / 15	512 / 15	1024 / 15	2048 / 15	4096 / 15	8192 / 15	256 / 15	512 / 15	1024 / 15	2048 / 15	4096 / 15	8192 / 15
CPU Intel i7 4770k	L_N1_M8_Gauss1	1.00x	1.48x	2.33x	6.43x	12.75x	36.21x	1.00x	1.60x	3.99x	7.61x	14.63x	30.48x
	H_N1_M3_Bennett5	1.00x	1.85x	3.57x	6.80x	12.51x	23.78x	1.00x	1.99x	3.93x	7.83x	15.55x	30.92x
AMP - GPU Nvidia Geforce 1080	L_N1_M8_Gauss1	1.00x	1.04x	1.07x	1.86x	3.11x	5.34x	1.00x	0.98x	1.08x	1.95x	2.98x	5.13x
	H_N1_M3_Bennett5	1.00x	0.97x	1.18x	1.27x	1.46x	1.85x	1.00x	0.96x	1.00x	1.05x	1.32x	2.59x
AMP - GPU Intel HD 4600	L_N1_M8_Gauss1	1.00x	1.13x	2.26x	4.53x	9.52x	18.85x						
	H_N1_M3_Bennett5	1.00x	1.03x	1.98x	3.94x	7.96x	16.01x						
AMP - WARP	L_N1_M8_Gauss1	1.00x	1.92x	3.70x	7.22x	14.39x	28.85x						
	H_N1_M3_Bennett5	1.00x	1.98x	3.87x	7.45x	15.12x	30.00x						

Table B.7: Population scaling of the PE approach for two test problems of different size. Each test was repeated 10 times for different parallelization methods, as well as for different population sizes and floating point precisions. The values specify the scaling of the computation cost in regards to the population size. They are color-coded from red (largest value) to green (smalles value).

Bibliography

[1] C. Gerlach, "Numerical Evolution – Using Genetic Hybrid-Methods to Solve Nonlinear Regression Problems," Master's thesis, Universität Hamburg, 2014.

[2] Committee on Publication Ethics, "Text Recycling Guidelines." `https://publicationethics.org/text-recycling-guidelines`, 2017.

[3] M. K. Transtrum, B. B. Machta, and J. P. Sethna, "Why are nonlinear fits to data so challenging?," *Physical Review E*, vol. 104, February 2010.

[4] L. Hogben, *Handbook of Linear Algebra*. CRC Press, second ed., 2014.

[5] G. A. F. Seber and C. J. Wild, *Nonlinear Regression*, pp. 4–10. Wiley, first ed., 1989.

[6] D. M. Bates and D. G. Watts, *Nonlinear Regression Analysis and Its Applications*. Wiley-Interscience, second ed., 2007.

[7] G. A. F. Seber and C. J. Wild, *Nonlinear Regression*. Wiley, first ed., 1989.

[8] G. A. F. Seber and C. J. Wild, *Nonlinear Regression*, pp. 15–18. Wiley, first ed., 1989.

[9] G. A. F. Seber and C. J. Wild, *Nonlinear Regression*, pp. 21–23. Wiley, first ed., 1989.

[10] M. K. Transtrum and J. P. Sethna, "Geodesic acceleration and the small-curvature approximation for nonlinear least squares." As article via Cornell University Library, article code:arXiv:1207.4999, July 2012. http://arxiv.org/abs/1207.4999.

[11] G. A. F. Seber and C. J. Wild, *Nonlinear Regression*, p. 621. Wiley, first ed., 1989.

[12] J. Holland, *Adaptation in Natural and Artificial Systems. Cambridge*. MIT Press, 1992.

[13] F. Giacobbo and M. Marseguerra, "Solving the inverse problem of parameter estimation by genetic algorithms," *Annals of Nuclear Energys*, vol. 29/8, May 2002.

[14] A. M. Kate Gregory, *C++ AMP: Accelerated Massive Parallelism with Microsoft Visual C++*. O'Reilly Media, Inc., first ed., 2012.

[15] Khronos Group, "The open standard for parallel programming of heterogeneous systems." `https://www.khronos.org/opencl/`, 2017.

[16] Nvidia Corporation, "CUDA." `http://www.nvidia.com/object/cuda_home_new.html`, 2017.

[17] Microsoft Corporation, "Direct3D." `https://msdn.microsoft.com/en-us/library/windows/desktop/hh309466(v=vs.85).aspx`, 2017.

[18] Microsoft Corporation, "C++ Accelerated Massive Parallelism." `https://msdn.microsoft.com/en-us/library/hh265137.aspx`, 2017.

[19] Khronos Group, "C++ Single-source Heterogeneous Programming for OpenCL." `https://www.khronos.org/sycl`, 2017.

[20] D. M. Bates and D. G. Watts, *Nonlinear Regression Analysis and Its Applications*. Wiley-Interscience, first ed., 1988.

[21] D. A. Ratkowsky, *Nonlinear Regression Modelling: A Unified Practical Approach*. Marcel Dekker Inc, first ed., 1989.

[22] J. Schnute and D. Fournier, "A new approach to lengthfrequency analysis: Growth structure," *Canadian Journal of Fisheries and Aquatic Sciences*, vol. 37(9), 1980.

[23] T. Blickle and L. Thiele, "A comparison of selection schemes used in genetic algorithms," tik report, 1995. Version 2.

[24] IEEE, "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2008*, pp. 1–70, Aug 2008.

[25] K. Levenberg, "A method for the solution of certain non-linear problems in least squares," *Quarterly of Applied Mathematics*, vol. 2, pp. 164–168, 1944.

[26] D. W. Marquardt, "An algorithm for least-squares estimation of nonlinear parameters," *Journal of the Society for Industrial and Applied Mathematics*, vol. 11/2, pp. 431–441, June 1963. http://www.jstor.org/stable/2098941.

[27] Various, "Wikipedia – numerical stability." `https://en.wikipedia.org/wiki/Numerical_stability`, 2017.

[28] Leslie Hogben, *Handbook of Linear Algebra*, ch. 58. CRC Press, second ed., 2014.

[29] J. Blinn, "Consider the lowly 2 x 2 matrix," *IEEE Computer Graphics and Applications*, vol. 16, pp. 82–88, March 1996.

[30] Microsoft Corporation, "restrict (C++ AMP)." `https://msdn.microsoft.com/en-us/library/hh388953.aspx`, 2017.

[31] National Institute of Standards and Technology (NIST), "Nonlinear Regression Statistical Reference Data Set." `http://www.itl.nist.gov/div898/strd/nls/nls_info.shtml`, 2003.

[32] Josef Tvrdik and Ivan Krivy, "Nonlinear Regression (14 difficult models)." `http://www1.osu.cz/~tvrdik/wp-content/uploads/nlmod14.pdf`, 2017.

[33] Nvidia Corporation, "GeForce GTX 1080 Whitepaper)." `http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf`, 2016.

[34] Various, "Toolbox for nonlinear regression in r: The package nlstools," *Journal of Statistical Software*, vol. 66, Aug 2015.

[35] Various, "R News," *The Newsletter of the R Project*, vol. 6, Aug 2006.

[36] M. K. Transtrum, B. B. Machta, and J. P. Sethna, "Geometry of nonlinear least squares with applications to sloppy models and optimization," *Physical Review E*, vol. 83, March 2011. citation code: Phys. Rev. E 83, 036701.

[37] Z. Drma and K. Veseli, "New fast and accurate jacobi svd algorithm. i," *SIAM Journal on Matrix Analysis and Applications*, vol. 29, no. 4, pp. 1322–1342, 2008.

Erklärung der Urheberschaft

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Masterstudiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel - insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen - benutzt habe. Alle Stellen, die wörtlich oder sinngem aus Verffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht

Ort, Datum Unterschrift