Hans-Martin Wulfmeyer

Genetic Programming for Automotive Modeling Applications

OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

INF FAKULTÄT FÜR INFORMATIK

Intelligent Cooperative Systems

Computational Intelligence

Genetic Programming for Automotive Modeling Applications

Bachelor Thesis

Hans-Martin Wulfmeyer

July 17, 2019

Supervisor: Prof. Dr.-Ing. habil. Sanaz Mostaghim

Advisor: Heiner Zille, M.Sc.

Advisor: Dr.-Ing. Markus Schori, IAV

Hans-Martin Wulfmeyer:
Genetic Programming for Automotive Modeling Applications
Otto-von-Guericke Universität
Intelligent Cooperative Systems
Computational Intelligence
Magdeburg, 2019.

Abstract

Genetic Programming (GP) is introduced as a symbolic regression technique for usage in automotive modeling applications. One use case is the modeling of the exhaust temperature in vehicles, of which two representative synthetic functions were obtained to examine and evaluate the capabilities of Symbolic Regression Genetic Programming.
The current state-of-the-art in GP was extensively examined and from these findings, suitable state-of-the-art techniques are used for an implementation of GP. The GP implementation created is then further used to conduct experiments for the two synthetic problems.
The results of GP compared to well established regression methods seem promising but also noticeably worse than for methods such as Gaussian Process Regression. While the results are noticeably worse than the methods compared to, GP produces an interpretable analytical function by using a quasi model-less method, which none of the methods compared to GP do. These can also give an insight into the general structure of the data used.

Contents

List of Figures

List of Tables

List of Acronyms

GP	Genetic Programming
EA	Evolutionary Algorithms
GPSR	Genetic Programming Symbolic Regression
ANN	Artificial Neural Networks
SVM	Support Vector Machines
RMSE	root mean squared error
MSE	mean squared error
SE	squared error
MAE	mean absolute error
LGP	Linear Genetic Programming
GE	Grammatical Evolution
CGP	Cartesian GP
EPLEX	ϵ-Lexicase
PGP	ParetoGP

1 Introduction and Motivation

Vehicle emissions and more efficient fuel consumption have been an ongoing issue and research topic in recent years with laws requiring lower levels of pollutants [1–3]. Heat transfer is directly related to engine efficiency and emission levels, which is why a focus is given to the exhaust temperature [1]. The high dependency on the exhaust temperature for these factors is also widely recognized and a simple but accurate model is crucial for improvement [3, 4]. Numerous engine, air system and injection system parameters, such as air flow, fuel injection quantity, engine speed, and rail pressure, affect the exhaust temperature [2, 3]. The relationship of these variables to the exhaust temperature is very often modeled with regression methods, for example by the Gaussian Process Regression Algorithm in the Bosch ECU software [2].
When using conventional regression methods a certain model is assumed beforehand, such as in linear, polynomial, and logistic regression, which try to fit their model to the data. The assumption is that the data is similar to the model used and if this is not the case there will not be an optimal solution. The luxury of having this a priori knowledge is often also not available so that deciding on a fitting model to our data proves to be difficult.
Most advanced supervised learning techniques, such as Artificial Neural Networks (ANN) or Support Vector Machines (SVM), also produce overly complex structures or even uninterpretable black boxes as solutions. Because of the nature of these black boxes, they are undesirable models for the regression task in this thesis, namely, exhaust temperature modeling. In this task the learned model is used as a predictor and having an interpretable solution is a highly desired property.
GP is one method among many in the field of Evolutionary Algorithms (EA) that employ the basic evolutionary rules of natural selection discovered by

Charles R. Darwin [5]. GP enjoys wide usage in the field of regression where it is employed as a method for symbolic regression known as Genetic Programming Symbolic Regression (GPSR). GPSR is a form of GP to “find a function, in symbolic form, that fits a given finite sampling of data points” [5]. It has been applied to and did comparatively well on many real-world problems including forecasting energy consumption from historical load electricity and weather information, forecasting the global mean temperature, modeling the Boston housing prices, predicting propylene concentration for chemical distillation towers and predicting soil water retention curves [6–9].
The approach in GP strays away from the mentioned regression methods in that it does not assume a model structure beforehand and that the solutions of the regression task are interpretable analytical functions [10]. In the light of these advantageous properties, it is being considered for exhaust temperature modeling in this thesis.

1.1 Goals

This thesis aims to answer what use GPSR has in the field of exhaust temperature modeling in automotive engineering. In cooperation with IAV Powertrain Mechatronics Research two synthetic regression problems respectively named f_1 and f_2 are provided, which are inspired by real-world data in the above mentioned field. In respect to these two problems this thesis aims to answer the following questions:

- What is the current state-of-the-art in GP?
- What parameter configurations for GP produce the best results?
- How does GP perform as a regression technique?
- How well does GP perform against established regressions methods?

To accurately measure the performance of GP several state-of-the-art approaches will be used, next to the standard GP implementation, for learning the models of synthetic regression problems. Models from conventional and advanced regression methods, such as Polynomial Regression, Multilayer Perceptron, and Gaussian Process Regression, will be used as a benchmark for comparison. Additionally, a hyperparameter search for selected parameters will be conducted to determine optimal configurations for GP.

1.2 Structure of Thesis

In Chapter 2, GP will be first introduced by explaining the fundamental knowledge necessary for this thesis, before giving an extensive overview of the current state-of-the-art in Chapter 3. Chapter 4 will explain the generation of the training data used and also give an overview of the implementation of GP used in this thesis. In the end, the chapter shortly focuses on different regression methods, which GP is compared to at the end of the following chapter. The results and the evaluation of the experiments are presented in Chapter 5, followed by Chapter 6, which contains the conclusion and future work to this thesis.

2 Fundamentals of Genetic Programming

GP is part of the larger class of EA, which are metaheuristics for numerical and combinatorial optimization problems for which no efficient solution algorithm is known. Kruse et al. define optimization problems as follows in [11].

Definition 2.1 (Optimization problem)**.** An *optimization problem* consists of the tuple (Ω, f) where Ω is the search space of potential solutions to the problem and $f : \Omega \to \mathbb{R}$ is a function that assigns a quality assessment $f(\omega)$ to each solution $\omega \in \Omega$.
An element $\omega \in \Omega$ is an **exact solution** for (Ω, f) if it is a global optimum.
An element $\omega \in \Omega$ is a **global optimum** of f if: $f(\omega') \preceq f(\omega)$[1] $\quad \forall \omega' \in \Omega$

EA do not guarantee optimal solutions and usually only provide approximate solutions to these problems as they merely constitute a guided random search. They are inspired by real-world biological evolution, that is Darwinian evolution. The basic assumption of EA is that the solutions ω will get better i.e. will have a better fitness, w.r.t f over time by mimicking the evolution process that is observed in the real-world. EA simulate this evolution process by creating a population and then selecting the solutions with the best fitness from the population. Genetic operators are applied to the selected solutions to create descendants, which form the new population, the next generation of solutions [11].
The basic algorithm underlying GP, which is very similar to EA, is shown in Figure 2.1. The GP algorithm depicted as pseudocode is displayed in Algorithm 2.1. The difference between EA and GP in the basic algorithm is that in

[1] $f(z) \preceq f(x) \quad \Leftrightarrow \quad x$ has a fitness better than or equal to z

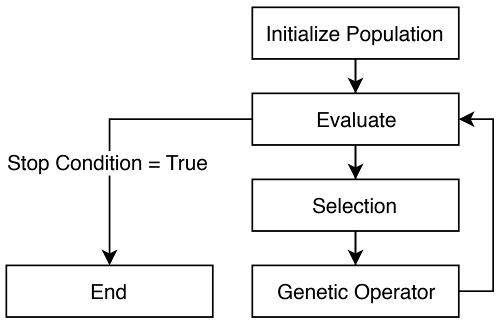


Figure 2.1: The basic GP process

the latter we only choose one genetic operator to be applied for every selected solution while in the former all genetic operators are applied one after another, mainly crossover and mutation, to each selected solution [5, 11, 12].
The solutions created by EA have a representation or alternatively called encoding, which determines how and in what way a solution is expressed internally. The representation is chosen specific to the problem at hand and is usually fixed in length [11]. It can be anything from a list of integers numbers to a simple bit array or more complex variants. If the representation of the solutions is of variable length, which aim to solve a specific task, it is called GP. Additionally, the solutions in GP consist of functions and terminals, which may include arithmetic operations, standard programming operations like loops and conditionals, mathematical functions, boolean functions, domain-specific functions, or variables and constants. The solutions in GP are usually also referred to as programs [5, 12].
In this thesis the specific task of the programs is to find the exact relationship between certain (real) numerical inputs to certain (real) numerical outputs by using symbolic regression, which is then called GPSR. In GPSR the goal is to find a model that describes that relationship as a function in an analytical form given a finite sampling of data points, the training data. An example of an analytical function is shown in Figure 2.2a. GPSR is also tasked with finding both the coefficients and the form of the function itself, in contrast to methods such as linear regression or polynomial regression where only the coefficients need to be found [10, 12].
For EA, including GP, it is imperative to define the building blocks the algo-

Algorithm 2.1 StandardGP

```
Input: $n$ = population size
Output: Population
 1: population ← init-population($n$)
 2: evaluate(population)
 3: while stopcondition ≠ True do
 4:     new_pop ← ∅
 5:     for n do
 6:         parent ← selection(population)
 7:         donor ← selection(population)
 8:         operator ← choose_genetic_operator
 9:         child ← operator(parent, donor)
10:         new_pop.append(child)
11:     population ← new_pop
12:     evaluate(population)
13: return population
```

rithm is composed of. In the following sections it is explained how solutions are represented in GP, how the initial population in the beginning can be created, how suitable solutions are evaluated and selected from the population and how and which genetic operators are applied to the selected solutions. It will also introduce all the necessary terminology needed and in particular this chapter and all the following chapters will have a focus on GPSR.

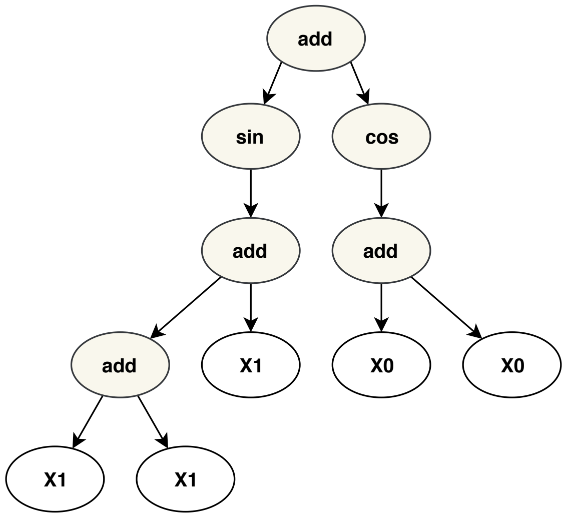
2.1 Representation

The representation of solutions, also called encoding, is an important issue because GPSR directly modifies the representation and it defines the set of all possible solutions to the problem, the search space. It also defines what kind of inputs are accepted, how they transform the inputs and produce an output. With that in mind, the representation directly impacts if GPSR is able to find a good solution or a useful solution at all [5, 11, 13].
One aspect of the representation in GPSR is the “physical” representation of the programs, i.e. how the computer displays the programs internally. Figure 2.2b shows the commonly used syntax tree. Another common variant is the prefix notation that directly corresponds to the syntax tree. It can be

incorporated into a list data type as shown in Figure 2.2c. Compared to a complex tree data structure a list data type is more computationally efficient and in most programming languages lists are also natively supported.
Because the prefix notation is simply another way of displaying a syntax tree, they can be used interchangeably in most cases, for example, when using initialization techniques or genetic operators [10, 12].

$$f(x_0, x_1) = \cos(x_0 + x_0) + \sin(x_1 + x_1 + x_1)$$

(a) analytical expression



(b) Syntax Tree

$+$
cos
$+$
x_0
x_0
sin
$+$
$+$
x_1
x_1
x_1

(c) List

Figure 2.2: Representations of analytical expressions

Because EA are inspired by biological evolution the representation is also called the “chromosome”, which consists of “genes”. A gene is one logical unit of a solution, which only partially determines a characteristic of an individual. The genes are usually the least minimal part of a representation, which are changed by the genetic operators [11].
In tree-based GPSR the ”genes“ are the elements in the function set $F = \{f_1, ..., f_n\}$ and the terminal set $T = \{t_1, ..., t_n\}$ with which the programs are build. The terminals represent the inputs into the program, which includes variables $(x_1, ..., x_n)$ and constant numbers. Usually, it also covers functions that take no inputs themselves, for example, a function that returns a ran-

dom number, which are called ephemeral constants. In contrast, all members of the function set take a specific number of arguments as input, which is called the arity. For GPSR the function set may consist of arithmetic operations $(+, -, *, \div)$, mathematical elementary functions $(sin, log, e^x, ...)$ and more elaborate functions $(max, min, abs, ...)$ [5, 11].
The function set is not only limited to the mentioned functions but is able to accept all that satisfy the closure property. This property asserts that all members of the function set are able to take the terminal set and all possible outputs of all members of the function set as input. Some examples where this is not the case is the division by or logarithm of zero and the square root or logarithm of a negative number because they are undefined operations or produce results that are not real numbers. The closure property is usually satisfied by defining protected versions, which handle the undesired behaviour and otherwise calculate the same as the original [5, 11, 13].
The terminal and function set need to satisfy another property, that is called completeness or sufficiency, which requires that it is possible to express a solution to the problem with the defined sets. In most cases the sets are over-sufficient, meaning that they contain more members than necessary because finding the smallest set necessary is usually an NP-hard problem [5, 11].

2.2 Initialization

In GP we randomly generate the individuals in the first generation of the population, like it is usually done in all other EA [12].
John R. Koza proposes two basic methods of generating the programs for GP, which are called "grow" and "full" [5]. They randomly create programs that do not exceed a declared maximum (syntax) tree depth, which is defined as the "longest nonbacktracking path from the root to an endpoint". Both methods generate the programs recursively beginning with the root node by selecting randomly from the terminal or function set.
The "full" method limits the selection to the function set until the specified depth is reached and then to the terminal set. This generates programs in which all paths from the root to the leaves have the maximum depth.

In contrast, the "grow" method does not limit the selection at all until reaching the maximum depth, which results in programs with variable depths and shapes, whereas the "full" method creates trees that are symmetric (if the arity of all functions were the same).
At last these two methods are incorporated in the "ramped half and half" method which creates half of the trees with the "grow" and the other half with the "full" method. Additionally, a minimum depth is defined and iteratively, for each depth between the minimum and maximum, an equal amount of trees is generated. This iterative process with increasing depth is referred to as "ramping". For example, if the minimum depth is 2 and the maximum depth is 6 the method will generate trees with the maximum depths 2, 3, 4, 5, 6 and for each depth, one half will be generated with the "full" and the "grow" method. This results in a great variety of sizes and shapes and thus a better initial population [5].

2.3 Fitness and Selection

The goal of the selection process is that individuals, which have a better fitness, are probabilistically more likely to be selected and with that more likely of producing more descendants [11]. For this, an appropriate fitness and selection mechanism has to be defined.

2.3.1 Fitness

The training data is the data set with which a model in supervised learning, which includes GPSR, is trained. It is made up of data points with one or more independent variables, the input, and usually only one dependent variable, the output. The fitness is always based on the error, which is calculated from input-output examples of the training data. It is determined by calculating the distance between the (scalar) output value y_i and the prediction of the output value $\hat{y}_i$ by the learned model from the input values $\vec{x_i}$. Usually, the fitness is then calculated by accumulating all distances for all data points from the training data. The fitness may also be standardized, normalized or

adjusted [5].
Two widely used fitness or model performance metrics for regression are the root mean squared error (RMSE) and the mean absolute error (MAE), which are displayed in Equations 2.1 and 2.2 respectively [14].

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \tag{2.1}$$

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{2.2}$$

The key difference between both is that the RMSE takes the squares and the MAE the absolute values of the errors. Both measures then sum the errors for all data points and calculate the mean value of the sum. Additionally, the RMSE takes the square root of the mean value. If the square root is missing it is just called mean squared error (MSE), if the square root and mean are omitted it is the squared error (SE).
Calculating the square of the errors results in the RMSE giving larger error values relatively more weight, in effect punishing variance in the error distribution and being more sensitive to outliers. The MAE, on the other hand, gives the same weight to all errors and would inadequately reflect larger errors, because of the influence of a large number of average errors. With these properties, the RMSE is superior at revealing differences in model performances and is more fitting for the use case in this thesis [14].
Another useful measure regularly used in regression is the coefficient of determination, denoted by R^2.

$$R^2 = 1 - \frac{\sum (y_i - \hat{y}_i)^2}{\sum (y_i - \bar{y})^2} \tag{2.3}$$

The R^2 is calculated by dividing the summed square errors of the evaluated model by the summed square errors of the “naive” model, the median $\bar{y}$ of the data. It is interpreted as the proportion of the variance in the data, which can be explained by the evaluated model. The value of R^2 usually ranges

from 1 to 0 with larger values denoting a better performance. It can also take negative values if the evaluated model has a worse performance than the "naive" model [15].
The R^2 and the RMSE are closely related to each other because they both utilize the SE. However, the SE is largely affected by the range of the y values in the data. The consequence is that depending on the data set a "low" or "high" RMSE does not necessarily relate to the exact performance of the model. Essentially the R^2 can be described as a relative-SE from which it is directly apparent how good a model is. This feature makes the R^2 useful for comparing regression results across different data sets or when dealing with data sets that contain noise.

2.3.2 Selection

When deciding on a selection mechanism the selection pressure property plays a decisive role, which describes how strong better fit individuals are preferred over the weaker ones [11, 12]. Selection pressure should be weak enough so that extraordinarily good programs can not completely dominate one generation and the diversity in the population is kept high. If there is very low to no selection pressure the selection is essentially random which is undesirable as well because the goal is to search for good fit individuals [11].
In GP any standard EA selection mechanism may be used, because the fitness definition is universally the same across most algorithms inspired by EA. However, the most commonly used methods in GP are tournament selection and Roulette-wheel selection [12].
In tournament selection k individuals are chosen at random from the population, which then partake in a tournament, where the one with the best fitness wins and gets to have a descendant in the next population. The size k of the tournament is chosen by the user beforehand and the k individuals that have been chosen are not excluded from the following tournaments. Because it produces only one winner per tournament at least n tournaments, for a new population of n individuals, have to be conducted [12]. The selection pressure on the population remains constant. All individuals have the same probabil-

ity to be chosen from the population and the fitness is essentially taken for a "ranking" of the individuals and the actual difference in fitness does not play any role at all [12].

In fitness proportionate selection methods, e.g. roulette-wheel selection, the fitness difference directly determines the likelihood of an individual to be selected. In roulette-wheel selection, the relative fitness is calculated for all individuals by dividing the fitness with the sum of the fitnesses of all individuals. The relative fitness is then taken as the probability for the individual to be selected. The fitness directly determines how likely it is for an individual to be selected, which results in a very high selection pressure and very good individuals to be able to completely dominate. This can lead to crowding, the population focusing only on a few regions in the search space, and then to premature convergence of the algorithm [11].

Oftentimes it can not be guaranteed that the best individual is selected at all or if selected will survive unchanged into the next generation. This is because the selection mechanism and the genetic operators, which are described in further detail in the following section, work with randomness. For that reason, elitism is often used, which copies the best or the best j individuals into the next population and guarantees that good solutions are not lost completely from detrimental random decisions [11].

2.4 Genetic Operators

Genetic operators are the means by which the algorithm navigates (locally) through the search space and advances the initial population, which has usually a very low fitness. The three primary operators used in GPSR are reproduction, crossover, and mutation and are applied based on assigned probabilities [13]. Only one genetic operator is applied to each selected solution, which means that the probabilities of all genetic operators have to sum up to 1.0.

Reproduction

Reproduction is the process of copying a selected individual into the next generation, which is done to prevent the algorithm from discarding good solutions [5].

Crossover

The (subtree) crossover operator takes two parents and combines them to form a child individual. This requires the selection mechanism to be performed twice to obtain the two parents.
The crossover operator randomly splits both parents into two separate trees by selecting a crossover point (node) in each, as displayed in Figure 2.3.

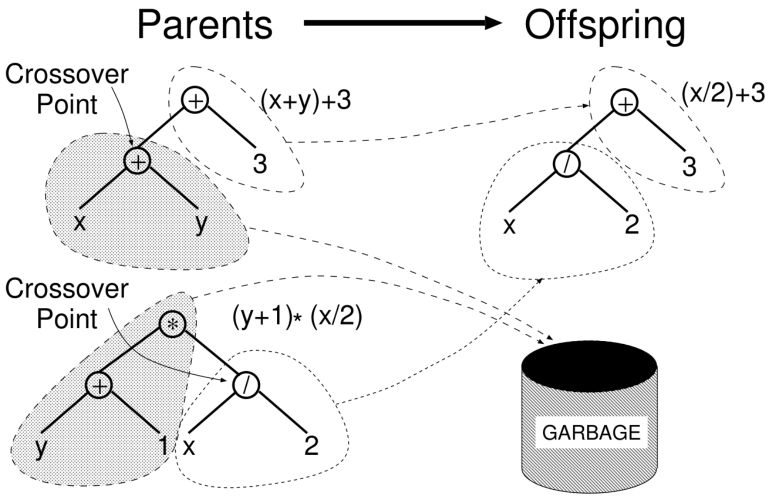


Figure 2.3: A crossover example in tree-based GP [12]

The subtree at the crossover point in the first parent is then replaced by the subtree at the crossover point in the second parent. This is essentially a restricted two-point crossover if the individual is represented in prefix notation, for which two fully working sub-expressions are exchanged. A subtree at the crossover point in subtree crossover is also always a fully working sub-expression. The crossover operator usually returns two children in EAs, which would also be achievable here, but which is not commonly done in GP [12].
One distinct advantage of returning one child instead of two is that it lends

itself better for parallelization because it removes the necessity of dynamically dealing with more than one individual being added.

Mutation

One goal of the mutation operator is to increase diversity in the population with the introduction of new genetic information into the population by performing random changes in individuals [11].
The mutation operators in GP can be distinguished as macro and micro mutation, which are used concurrently. While micro mutation only changes one gene (for tree-based GP terminals or functions) or even only a part of a gene from the program, a macro mutation operator changes entire sequences of genes at once [16]. The mutation operators commonly used for that are subtree mutation and point mutation [12].
Subtree mutation randomly selects a mutation point (node) in the selected parent and replaces the subtree at the mutation point with a randomly generated subtree, which is similar to the way it is done in crossover [13]. This is displayed in Figure 2.4.

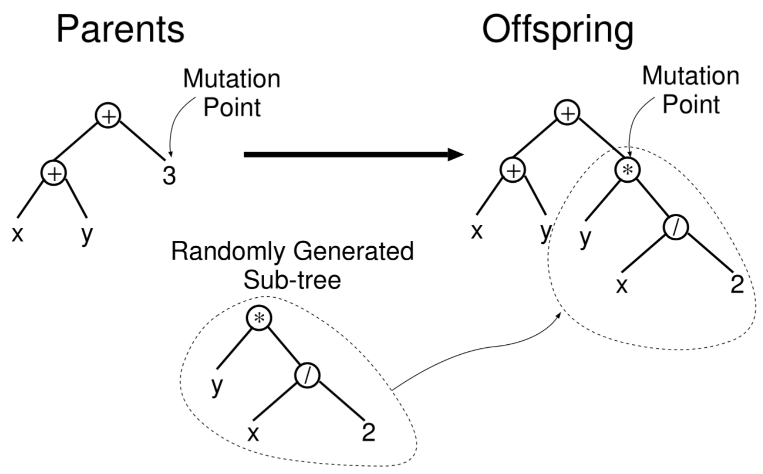


Figure 2.4: A subtree mutation example in tree-based GP [12]

Subtree mutation can be achieved by performing subtree crossover with a randomly generated individual from the initialization techniques explained in Section 2.2. This is referred to as the "headless chicken" crossover [17].

Point mutation, also called standard mutation, is a generalization of bit(-flip) mutation. In point mutation each node in an individual has the same probability, defined by the user beforehand, to be mutated i.e. replaced by a randomly generated node. That is this results in multiple points being mutated separately [11, 12]. To prevent the generation of useless individuals, point mutation replaces terminals with random terminals and functions with random functions of the same arity [12].

2.5 Bloat

One of the greater issues that GP faces in practice is that solutions can grow arbitrarily large without gaining a significant amount of fitness, which is referred to as bloat [12]. Bloat has several disadvantages which include overfitting and performance issues in memory and computation [18]. The solutions can even grow to a size for which it gets impossible to evaluate them.

Bloat is partly caused by introns, which are non-effective parts of a solution [11]. Non-effective means that they do not have any effect on the fitness of the solution. The multiplication by or division by 1 is a simple example of an intron in GPSR.

Two widely used basic bloat prevention methods are limiting the tree depth or tree length to a certain size and the parsimony pressure method [12]. The parsimony pressure method multiplies the size of the solution with the parsimony coefficient, usually a constant, and then adds the resulting value to the fitness. The fitness of the individuals are penalized for the size. Larger programs will have a greater penalized value, which leads to the effect that smaller programs are preferred over larger programs with a similar raw fitness in the selection process. More advanced methods are, for example, to use multi-objective optimization with the fitness and the size of the solutions or to introduce genetic operators that replace subtrees in an individual with smaller subtrees, similar to subtree crossover. Methods to reduce or prevent bloat are more extensively discussed in Chapter 3.

3 State-of-the-Art

In this chapter current findings and algorithms in GP will be introduced.
Most techniques from EA take a modular approach to the components in the algorithm, such as selection methods or genetic operators, and GP is no different in that regard [5, 11].
Especially selection methods are highly modular as they are mostly based only on the fitness and are indifferent to, for example, the encoding. However genetic operators and the initialization of solutions are highly customized to the encoding, for example, the subtree crossover described in Section 2.4, that works exclusively on tree-structures or representations derived from tree-structures [12].
This becomes an issue when the goal is to utilize these methods for different representations than they were originally invented for, if even possible. One solution can be to translate these methods for usage on other encodings. It might even be feasible to transform solutions temporarily into another encoding to make genetic operators or initialization methods usable.
This is why each new representation tends to have their own version of crossover and mutation, as will be evident from Section 3.1 where different GP representations are explained.

3.1 Representation

Over the years different representations for GP have been proposed in scientific literature and research. One motivation behind using different representations is that they open up certain problem domains that might be more suitable

than others [16].
In 2012 a survey was conducted by D. R. White et al. in the Genetic Programming research community [19]. Among other questions, the authors asked the participants what type of GP they use and which one they use the most. The results concluded that standard GP (i.e. tree-based GP) is still the most widely used type of GP followed by Grammatical GP and then “Standard GP with strong typing or other modifications”.
Strong typing refers to the practice to define “rules” for the terminal and function nodes determining which connections they are allowed to create with other nodes. One application is for the usage of more than one data type in the function and terminal set e.g. booleans and floats. With certain rules, strong typing guarantees that the closure property is adhered to [12].
Other types of GP in the survey question include Stack-based, Linear and Cartesian GP, which are briefly explained in the following sections.

3.1.1 Linear

In their book on LGP Brameier and Banzhaf concentrate extensively on the fundamental aspects of linear program representation [16].
Figure 3.1 is a simple example of an LGP program, which calculates the same analytical function displayed in Figure 2.2a.

```c
double x[2];
double r[8];
// r[7] = r[0] - 59;
r[4] = x[0] + x[0];
// r[2] = r[5] / r[4];
r[6] = x[1];
r[7] = r[6] * 3;
// r[5] = x[1] + 15;
r[0] = sin(r[7]);
r[5] = cos(r[4]);
r[0] = r[0] + r[5];
```

Figure 3.1: Example LGP program in the C-programming language calculating $\cos(x_0 + x_0) + \sin(x_1 * 3)$

Linear does not refer to the program's ability to only solve linear problems but "to the structure of the program representation" [16]. LGP is inspired by machine language instructions, which use registers to operate on and save results.
One principle in LGP is that registers can arbitrarily often be reused for instructions or as storage. This causes that certain instructions will be structural introns, which are distinguished from semantic introns. Introns are non-effective parts of an individual.
Structural introns do not cause any manipulations of the end result whereas semantic introns are structurally effective but are still non-effective regarding the fitness. Structural introns occur either because the result register is overwritten by another instruction, it is not used at all or it is not used for further instructions that lead to manipulations of the end result.
Semantic introns also occur in tree-based GPSR, while structural introns are non-existent because of the nature of syntax trees in which all parts necessarily affect the end result in some way. However, if booleans and the corresponding functions are included in the function and terminal set, which are usually not included in GPSR, structural introns can occur as well.
Structural introns can easily be identified and removed in LGP, which is only done for the evaluation because introns are viewed as an integral part.
The example in Figure 3.1 only contains structural introns, which are commented out. Denoted by r are the calculation registers that are freely usable by the program and whose size has to be defined beforehand by the user. In addition to that, there are write-protected registers denoted by x, which hold the program inputs. The standard output or result register is $r[0]$. The internal representation of an individual is usually not in C code but is translated into C code, or any other high-level programming language, for the purpose of executing the programs. For the genetic operators each instruction, that is

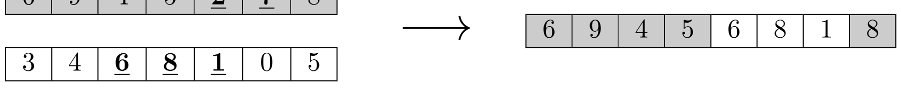


Figure 3.2: Example of two-point crossover, the numbers 2,7 are replaced with 6,8,1 from the other individual.

each line in the example in Figure 3.1, is interpreted as one gene in the individual. The mutation operators then either insert or delete a single instruction or treat the instructions as atomic and merely change a single component of an instruction [16].
One crossover technique utilized in LGP is similar to the traditional two-point crossover, which in its adapted version replaces a randomly chosen continuous sequence of instructions from another individual with one in the chosen individual. In Figure 3.2 an example of the adapted two-point crossover with integer numbers is displayed. The motivation for using LGP is that tree-based GP requires an interpreter that translates the individuals, usually represented in a tree-shape or prefix-notation, into executable code. In contrast, LGP uses machine code, which provides some speed-up because it is directly executable [12].
Markus F. Brameier and Wolfgang Banzhaf also argue that because of its weaker constraints LGP has a smaller variation step size, that it produces more compact solutions, which may also be executed more efficiently than tree-based GP [16].

3.1.2 Stack-based

One of the defining works regarding stack-based GP is the Push programming language and the resulting genetic programming system PushGP by Spector and Robinson [20].
The fundamental of stack-based GP is the usage of global data stacks. A stack is a type of data container that follows the LIFO (last-in first-out) principle and uses the two operators push to insert and pop to remove elements at the top of the stack. The stacks are filled with arguments, which are then passed to instructions. The results from the instructions are then again pushed onto the stack. This leads to the usage of the postfix syntax for the representation of individuals. In the postfix syntax, the arguments are located before the instructions by which they are used. The concept is similar to the representation of individuals in prefix notation for the usage in list data types, for example in Figure 2.2c. In prefix notation, the list in Figure 2.2c is evaluated in a recur-

sive manner from top to bottom, while in stack-based GP the postfix notation it is evaluated from bottom to top without the need of recursion.
Figure 3.3 is a simple example of the representation of `(2+4)*3` in postfix notation. It is read from left to right.

```
3 2 4 + *
```

Figure 3.3: Example of `(2+4)*3` in postfix notation

In stack-based GP the three numbers `(3,2,4)` in Figure 3.3 are first pushed onto the empty stack and then the `+` instruction is executed by taking (popping) the top two numbers `(2,4)` from the stack. The resulting value of `6` is then pushed onto the stack and the `*` instruction pops the top two numbers `(3,6)` from the stack. In the end, the stack only contains the number `18` and no instructions are left, which means the evaluation is done and the value left is the result.
The evaluation process also causes that the same calculation can be expressed in multiple ways. For example Figure 3.4 results in the same calculation as the example in Figure 3.3.

```
2 4 + 3 *
```

Figure 3.4: Another example of `(2+4)*3` in postfix notation

In PushGP it is also possible to use multiple data types in which case each data type is provided with a separate stack. The instructions then take the arguments from whichever stack they are needed from.
Stack-based GP, in particular, the push programming language, also implements list and code-manipulation instructions, which among others includes instruction for the copying of entire sub-expression. These instructions allow for recursions, loops and self-modifying programs to be generated. For the usage of variables the two instructions `GET` and `SET` are provided.
If an instruction is executed that can not be provided with sufficient arguments from the stack it is simply ignored and discarded. This is why the genetic operators in stack-based GP do not have to be restricted to only generate offspring

that would be valid programs or expressions themselves, in contrast to tree-based GP. Because of this, the genetic operators act without restrictions on the postfix expression of the programs and traditional crossover and mutation operators can be applied.
One interesting development regarding PushGP is the addition of epigenetic information into the individual [21]. Each individual has a corresponding boolean vector of the same length, which is referred to as the epigenome, that defines if an element is active. If an element is inactive it is ignored when the program is executed.

3.1.3 Grammatical

Grammatical GP is a method to enforce user specified structures on the solutions by defining building rules similar to strong typing. This is realized by defining a generative grammar, which the individuals have to adhere to [22]. An example for such a grammar in Backus Naur Form (BNF) notation is displayed in Equation 3.1 [12].

$$\begin{aligned} \textit{tree} &::= E \times \sin(E \times E) \\ E &::= \texttt{var} \mid (E \ \texttt{op} \ E) \\ \texttt{op} &::= + \mid - \mid \times \mid \div \\ \texttt{var} &::= x \mid y \mid z \end{aligned} \tag{3.1}$$

Each line in a BNF grammar is a production rule, which define the general structure of the programs. On the left-hand-side are the non-terminal symbols while all non-rewriteable symbols $(x, y, z, +, -, \times, \div)$, which only appear on the right-hand-side of the equation, are terminal symbols. The production rules are then used recursively to build individuals, beginning with the start rule *tree* [12, 22].
In Equation 3.2 an example individual was build by randomly evaluating the production rules from Equation 3.1 from top to bottom. Each production rule was iteratively evaluated until no non-terminal symbols of the specific

production rule were left.

$$\begin{aligned}
tree &::= E \times \sin(E \times E) \quad (3.2)\\
tree &::= \texttt{var} \times \sin((E \ \texttt{op} \ E) \times \texttt{var})\\
tree &::= \texttt{var} \times \sin((\texttt{var} \ \texttt{op} \ (E \ \texttt{op} \ E)) \times \texttt{var})\\
tree &::= \texttt{var} \times \sin((\texttt{var} \ \texttt{op} \ (\texttt{var} \ \texttt{op} \ \texttt{var})) \times \texttt{var})\\
tree &::= \texttt{var} \times \sin((\texttt{var} - (\texttt{var} \div \texttt{var})) \times \texttt{var})\\
tree &::= x \times \sin((z - (y \div x)) \times y)
\end{aligned}$$

The concept of using grammars for the generative building of individuals for GP is used in Grammatical Evolution (GE) by O'Neill and Ryan [22]. The individuals in GE are represented as binary strings of variable length (multiples of 8) and a consecutive group of 8 bits is read from the individual and converted to an integer. This results in an individual of several integers.
Beginning with the start rule the building process in GE works in a similar way as in Equation 3.2. But, instead of making a random choice for the production rules, like done above in Equation 3.2, the integer numbers are mapped to one of the options of a production rule with the mapping function in Equation 3.3.

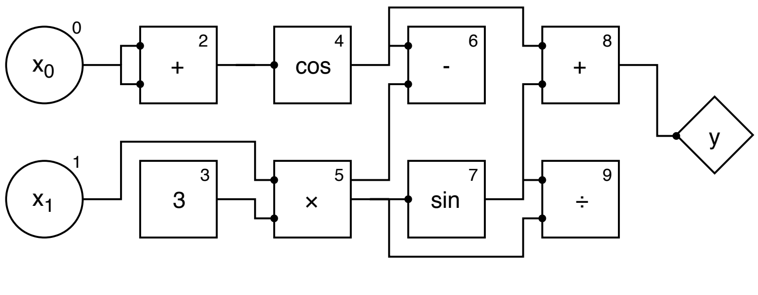
$$\begin{aligned}
option &= n \ \texttt{MOD} \ m \quad (3.3)\\
n &= \text{integer number}\\
m &= \text{number of options for production rule}
\end{aligned}$$

For example, the rule E in Equation 3.1 has two possible options, which would have the assigned numbers 0 and 1.
The building process begins anew from the left side at each step of the evaluation, that is after mapping a rule the first non-terminal symbol beginning from the left side is being evaluated next.
There is the possibility that the number of integers runs out in which case the individual is wrapped from the end to the start and we reuse the already used integers. With wrapping an individual could always apply the same option repeatedly resulting in an endless loop, which is why there is a wrapping limit.

By reaching the wrapping limit an individual is checked for validness and if flagged as invalid is assigned the lowest possible fitness.
One major advantage of Grammatical GP is that domain knowledge can easily be expressed by creating certain production rules. Additionally, the binary string representation in GE allows the easy application of several various genetic operators commonly used in EA [23].

3.1.4 Cartesian

Tree-structures are special types of graphs, which is why several researchers have looked into using other graph types for the development of individuals in GP [12]. One such method is CGP, which uses directed acyclic graphs (DAG) to represent the individuals [24]. As they are used in GP the nodes in a DAG can have none, one or more parent nodes while in a tree each node has exactly one parent.
In CGP the internal representation is a list of integers or the corresponding string of bits. The integers are then used in a mapping function to implement the individuals as two-dimensional grids of computational nodes with rows and columns, which is why it is called Cartesian. One example of an individual in CGP with the integer and graph representation is shown in Figure 3.5.



100 610 521 213 045 452 147 375 8

Figure 3.5: Example CGP graph calculating $\cos(x_0 + x_0) + \sin(x_1 * 3)$ with the corresponding integer representation using the function set $(-_0, +_1, \times_2, \div_3, \sin_4, \cos_5, rand_6)$

The mapping function treats a three consecutive integers as one node with the first number determining what the type of the node is by using a function look-up table. The other numbers represent the addresses of nodes or program inputs. These numbers define where the current node gets its inputs from and it is assumed that each function takes as many inputs as the function with the maximum arity. Not needed inputs are then simply ignored. The graphs in CGP are directed and feed-forward, i.e. the nodes are only able to get their inputs from any of the previous columns of nodes.
Depending on how many outputs an individual needs there are output nodes with corresponding integers in the internal representation, which determine the addresses of the nodes where the outputs are from.
The graphs in CGP have maximum sizes for the columns and rows, which are determined by the user. Additionally, there is a user defined parameter that takes the positive natural numbers. The value of this parameter determines the number of directly preceding columns from which a node can take the outputs as inputs, in addition to the program inputs themselves. This is referred to as the connectivity of the graph.
Besides a simple point mutation for any of the integers in the representation, crossover is not used much in the original CGP because it has shown to be disruptive and negatively affecting the convergence rate, while not improving the general performance. A new method for crossover from Clegg, Walker, and Miller has shown to not be disruptive and generate a faster convergence in the evolutionary process [25].
For this new crossover operator the individuals are internally represented with floating point numbers in the range of [0,1] instead of integers, however, the floating point representation is still transformed into the integer-based representation. The transformation is done with the mapping function in Equation 3.4.

$$\begin{aligned} & floor(f * k) \qquad (3.4) \\ & \quad f = \text{floating point number} \\ & \quad k = \text{total number of functions or } num_{inputs} + num_{currentnode} \end{aligned}$$

Depending on if a mapping to the available functions is needed or for the inputs of a node, k is chosen to be either the total number of functions available or the sum of the total number of inputs and the index number of the current node.
An offspring between two parents p_1, p_2 in floating point representation is then generated pointwise with Equation 3.5, where r denotes a per offspring uniformly generated random number.

$$\text{offspring} = (1 - r) * p_1 + r * p_2 \tag{3.5}$$

Another recent development is self-modifying CGP (SMCGP), which builds on top of the original CGP and includes similar self-modification functions found in LGP, such as adding, copying and deletion of graph nodes. The modifications are not applied immediately but are build as functions into the graph representation of the program. Each time the graph is evaluated with input data the reached self-modifying functions are added to a list of pending manipulations and applied between runs [26].
One advantage of using graphs is that it allows the implicit reusing of partial results of an individual. On top of that the graphs in CGP can have an arbitrary number of outputs [24].

3.2 Semantic Crossover and Mutation

As has already been stated in Section 2.4, genetic operators are the search operators for the development of better individuals in the population. The genetic operators have a strong influence on the success of GP, which is why one particular research focus is on improving the genetic operators [5, 13, 27, 28].
One interesting development is the utilization of the program semantics in the genetic operators named semantically driven crossover and mutation [29, 30]. The semantic is the meaning or the behavior caused by a GP program, i.e. the output values. The syntax is the representation of the program.
Both operators have been applied to boolean GP programs. This was done

by utilizing Reduced Ordered Binary Decision Diagrams (ROBDD), which are boolean functions transformed into directed acyclic graphs [31]. If the offspring created by crossover or mutation produces the same ROBDD as the parents they are declared semantically equivalent and discarded. Instead of adding the offspring the original parents are then added to the next generation. Both crossover and mutation create offspring that are semantically different to their parents.
Semantically driven crossover tries to improve the general fitness performance and the efficiency regarding bloat of the original crossover [29]. Semantically driven mutation aims to just improve the performance of the mutation [30].
Semantically driven crossover and mutation are based on trial and error and do not give any insight into the relationship of the syntax and the semantic of the parents and offspring. Instead of being directly able to search the semantic space produced by the programs they rely on pure coincidence. This is why Moraglio, Krawiec, and Johnson created geometric semantic operators, which directly search the semantic space by creating offspring, which have a certain semantic distance to the parents [32].
For the purpose of this thesis only the implementations for real analytic programs are being considered, which are given in Definition 3.1 and 3.2.

Definition 3.1 (Geometric Semantic Crossover)**.** For the two parents $T_1, T_2 : \mathbb{R}^n \to \mathbb{R}$, the crossover operator builds the offspring
$T_C = (T_1 \times T_R) + (T_2 \times (1 - T_R))$ where T_R is either a random value in $[0, 1]$ or a random program with codomain $[0, 1]$.

Definition 3.2 (Geometric Semantic Mutation)**.** For the parent $T : \mathbb{R}^n \to \mathbb{R}$, the mutation operator builds the offspring
$T_M = T + ms \times (T_{R1} - T_{R2})$ where T_{R1} and T_{R2} are random programs and ms is the mutation step, a real valued parameter in $[0, 1]$.

The operators create a new individual by manually inserting already existing individuals into the given form. For example, when using syntax trees as representation the resulting individual will be a syntax tree as well, which will contain the calculation from the definition in syntax tree form with the parents inserted as sub-trees.

Both operators are utilizable for all program representations that permit the creation of new individuals with the processes in Definition 3.1 and 3.2.
One drawback of the geometric semantic operators is the significant growth in the offspring because they are created by including both parents, when applying crossover, or two random functions when considering mutation. This problem was addressed by utilizing function simplifiers in the GP implementation of Moraglio, Krawiec, and Johnson. These could also be used without any repercussions because the program syntax does not matter for the operators. However, function simplifiers require additional computational performance. Additionally to the function simplifiers, to keep the growth even lower, the crossover operator was implemented using random real values instead of random programs.
In their experiments, it was shown that geometric semantic crossover and mutation produce considerably better symbolic regression results than the traditional operators on several random polynomial functions ranging from degree 3 to 10 [32].

3.3 Selection

The following sections will introduce the ϵ-Lexicase (EPLEX) selection mechanism, multi-objective selection in general as well as one multi-objective selection mechanism named ParetoGP. Furthermore, a short introduction into complexity measures for the programs in GP will be given.

3.3.1 ϵ-Lexicase Selection

EPLEX is a parent selection mechanism based on lexicase selection by Lee Spector [33]. Lexicase selection treats each data point as a separate fitness case on which the solutions can perform on. A parent is then selected by filtering the pool of parents also called candidates based on how they perform on each data point separately. The filtering is done with a pass condition, which automatically filters all candidates from the pool that do not meet the required criterion. The algorithm described by Lee Spector is one of the simpler variants of lexicase selection named “global pool, uniform random sequence, elitist lexicase selection” or just short lexicase selection [33]. The algorithm is displayed in Algorithm 3.2 as pseudocode.

Algorithm 3.2 lexicase selection algorithm

```
1: procedure LEXICASE(population, data)
2:     ppool ← population
3:     data ← random_order(data)
4:     while size(ppool) $\neq$ 1 $\wedge$ data $\neq \varnothing$ do
5:         date ← pop(data)                ▷ pop returns and removes the first point
6:         ppool ← Filter(ppool, date)     ▷ Filtering by applying the pass condition p
7:     return random_choice(ppool)
```

The pool of potential candidates is constructed as the entire population. For every parent selection scenario, the data points are ordered uniformly at random and applied by the candidates in that order. For each data point only the candidates that perform as well as the best candidate are retained in the pool, which is given as a pass condition in Equation 3.6, which is then used in a filter mechanism.

$$\mathrm{p}_\mathrm{i}^t = \mathrm{IF}[f_i^t \leq f_*^t] \tag{3.6}$$

The pool of candidates is filtered on the data points until there is only one candidate left or no other data points are left. If there are no data points left and there is more than one candidate left a random candidate from the pool is returned.
Because the data points are randomized at each selection step the resulting parents are pressured to perform well on unique combinations of data points leading to an increased diversity [34].
Variants of lexicase selection can be created by altering how the pool of potential parents is created, how these are then filtered and lastly how the different data points are applied [33].
The basic Lexicase selection Algorithm shown in Algorithm 3.2 works well on training data with discrete errors but does not do well in continues-valued problems. In these problem domains, individuals are unlikely to have the same

fitness, which would result in all individuals being filtered at once. This is why La Cava, Spector, and Danai proposed an addition to the algorithm resulting in the derivative Epsilon-Lexicase selection [8].
In Epsilon-Lexicase selection, filtering is applied, which is less strict by introducing an epsilon value as an absolute threshold in which the candidates are not being filtered. Two variants of the pass condition p_t are denoted as follows:

$$\mathrm{p_i}^t = \mathrm{IF}[f_i^t < f_*^t + \epsilon] \tag{3.7}$$

$$\mathrm{p_i}^t = \mathrm{IF}[f_i^t < \epsilon)] \tag{3.8}$$

In Equation 3.7 an individual or candidate i does pass on a specific data point t if its fitness f_i^t on the data point is less than the sum of the best fitness f_*^t of all candidates on the data point and ϵ. Equation 3.8 is similar but the best fitness f_*^t is omitted. The pass condition returns *true* for passing the respective data point and the result is utilized by a filter, which removes the respective candidates that do not pass from the pool.
The pass conditions in Equations 3.7 and 3.8 have the disadvantage that ϵ has to be defined by the user, that the optimal ϵ value varies greatly for different problems and a static ϵ value does not provide a continuing selection pressure. These disadvantages are largely overcome by the dynamic variants in Equations 3.8 and 3.10, which include the median absolute deviation (MAD) of f^t (the median of the absolute deviations from the median of the fitnesses of all candidates on data point t) [8].

$$\epsilon(f^t) = \mathrm{MAD}(f^t) = \mathrm{median}(|f^t - \mathrm{median}(f^t)|) \tag{3.9}$$

$$\mathrm{p_i}^t = \mathrm{IF}[f_i^t < f_*^t + \epsilon(f^t)] \tag{3.10}$$

$$\mathrm{p_i}^t = \mathrm{IF}[f_i^t < \epsilon(f^t)] \tag{3.11}$$

The pass conditions in Equations 3.7 and 3.10 define the passing in relation to the best fitness while the pass conditions in Equations 3.8 and 3.11 do not, which results in the effect that for the latter no parents might survive at all.
La Cava et al. differentiate between three different versions namely static, semi-dynamic and dynamic ϵ-lexicase selection, which all use Equation 3.10.

Static ϵ-Lexicase determines the best fitness f_*^t and $\epsilon(f^t)$ once per generation, in essence, it ignores the pool of remaining candidates. In contrast to that semi-dynamic determines f_*^t from the remaining candidates that were not yet filtered. Dynamic ϵ-Lexicase selection determines both f_t^* and $\epsilon(f^t)$ from the remaining candidates. In their experiments semi-dynamic and dynamic ϵ-Lexicase performed best and La Cava et al. suggest to use the semi-dynamic variant because its results show the lowest mean test ranking and the median number of data points used per selection is higher than in the purely dynamic version [35].

3.3.2 Multi-objective Selection

As explained in Section 2.5 bloat is an issue that needs to be controlled. Instead of using the parsimony pressure method, which is a linear relationship between the complexity of the solution and the fitness, this issue may be redefined as a multi-objective problem. If the complexity measure is defined as the second objective criterion, which is then minimized concurrently with the fitness of the solutions, it can be solved as a multi-objective problem. This approach removes the necessity to define the parsimony coefficient and it provides a more efficient search for feasible solutions while simultaneously reducing bloat.
Multi-objective selection is usually by means of Pareto-dominance, which is based on Pareto-optimality as defined in Definition 3.3 [11].

Definition 3.3 (Pareto-optimality)**.** An element s is called Pareto-optimal w.r.t. the objective functions $f_i, i = 1, ..., k$ if there does not exist any element s' in the search space Ω for which the following two properties hold:

$$f_i(s') \preceq f_i(s) \qquad \forall i \;\; 1 \leq i \leq k \tag{3.12}$$

$$f_i(s') \prec f_i(s) \qquad \exists i \;\; 1 \leq i \leq k \tag{3.13}$$

$f(z) \preceq f(x) \;\; \Leftrightarrow \;\; x$ has a fitness better than or equal to z

While Pareto-optimality is defined over the whole search space, Pareto-dominance only considers the currently known solutions i.e. the population in EA. An element s is then considered non-dominated if there does not exist

any other element s' in the current population for which the Equations 3.12 and 3.13 hold.
There are some theoretical and practical problems with traditional multi-objective optimization regarding the usage in GP. One issue is that individuals with the same objective properties are treated as non-dominated. This results in identical or even just very similar individuals all being declared non-dominated, which causes a severe lack of diversity in the population. Another issue is the assumption that the two objectives, fitness and complexity, are considered equal regarding the goodness of the solution when the actual goal is to create individuals with better fitness. This generally causes a shift to smaller individuals while neglecting the fitness because a small size is easier and faster to achieve than a better fitness [36].
These issues were also encountered by Kommenda et al. when they employed the Non-dominated Sorting Genetic Algorithm-II (NSGA-II) in GPSR. Without any adaptions, the NSGA-II algorithm resulted in the GP population made up entirely of individuals with only one node and GP not being able to evolve larger more complex individuals with a better fitness [37].
Strength Pareto Evolutionary Algorithm 2 (SPEA-2), an evolutionary multi-objective optimization algorithm, faced similar issues and has been used in GP by making adaptions to the base algorithm [38].
One general adaption to multi-objective optimization for complexity reduction in GP should be diversity control. One way to achieve this is by treating individuals with the same objective properties as dominated and to only keep one of them [36]. To facilitate this effect even further Kommenda et al. discretized the fitness values in GPSR by rounding to a fixed number of decimal places. This causes that individuals that only differ in a very small neglectable amount are still interpreted as equal regarding the fitness [37].

ParetoGP

ParetoGP is a multi-objective optimization algorithm for GP to reduce the complexity of individuals by utilizing the non-dominated individuals of the population of programs [39].

In ParetoGP the non-dominated programs are saved in a cross-generational archive, which is updated after each evolutionary iteration in GP. The conventional population is created by choosing parents from the archive and the previous generation's population. The parents from the archive are chosen at random because they are all considered equally as good. Another parent is only necessary in the case of crossover, which is then chosen from the conventional population with traditional selection mechanisms, e.g. tournament selection based on fitness. Between the selected parents', the traditional crossover is then performed. From the newly created population and the current archive, all non-dominated solutions are selected and saved as the new archive.
The basic ParetoGP algorithm is outlined in Algorithm 3.3 as pseudocode and only differs from the StandardGP algorithm in Algorithm 2.1 in Lines 3, 7, 14 and 15. Lines 3 and 14 are newly added, while in line 7 instead of selecting from the population we randomly select from the archive and in line 15 the archive is returned instead of the population.

Algorithm 3.3 ParetoGP

```
Input: $n$ = population size
Output: Archive
 1: population ← init-population(n)
 2: evaluate(population)
 3: archive ← return-non-dominated(population)
 4: while stopcondition $\neq$ True do
 5:     new_pop ← $\varnothing$
 6:     for n do
 7:         parent ← random_selection(archive)
 8:         donor ← selection(population)              ▷ only based on fitness
 9:         operator ← choose_genetic_operator         ▷ Crossover / Mutation
10:         child ← operator(parent, donor)
11:         new_pop ← new_pop $\cup$ child
12:     population ← new_pop
13:     evaluate(population)
14:     archive ← return-non-dominated(archive $\cup$ population)
15: return archive
```

Complexity Measures

In Section 2.3.1 fitness measures were extensively talked about but deciding on a fitting complexity measure for the multi-objective approach is more problematic. For example, complexity may be defined from the features of the programs, such as tree-height, length, tree-depth. The issue is that these only consider the form of the programs, i.e. the representation of the individuals and not the mathematical complexity of the solution. Two different programs with one only containing additions while the other only uses multiplications can have the same amount of nodes, tree-height or length. But they would be very dissimilar in their actual mathematical complexity. This problem highlights that it might be beneficial to take the mathematical complexity of the solutions into account to determine which programs are more complex.
Vladislavleva, Smits, and den Hertog applied ParetoGP with the addition of the novel complexity measure “Order of Nonlinearity”, which accounts for the non-linearity of the solutions presented [40]. The “Order of Nonlinearity” recursively defines the complexity by accumulating the complexity of all sub-trees. The degree of a minimal Chebyshev approximation is included in several steps as well.
Chebyshev approximations are polynomials that follow certain properties and are called approximations if they reproduce the behavior of a given function to a certain precision in a defined interval or set of input values. A Chebyshev approximation is then declared “minimal” if it has the smallest polynomial degree possible [40, 41].
Finding a minimal Chebyshev approximation includes the computation of the approximation error for several Chebyshev polynomials of certain degrees. For each individual, this is also repeated numerous times because they are needed for several sub-expressions. This is why Kommenda et al. derived a new complexity measure from the “Order of Nonlinearity” without using Chebyshev approximations, which results in lower computational cost and still captures the complexity sufficiently well [37]. This complexity measure is presented in Definition 3.4.

Definition 3.4 (Kommenda Complexity)**.** For a node n with $op \in (terminals, functions)$ and the possible child nodes n_1, n_2, the complexity $\text{comp}(n)$ is determined by the class of *op*:

$$\begin{aligned}
&op = \text{constant} && \rightarrow && 1 \\
&op = \text{variable} && \rightarrow && 2 \\
&op \in (+, -) && \rightarrow && \text{comp}(n_1) + \text{comp}(n_2) \\
&op \in (\times, \div) && \rightarrow && (\text{comp}(n_1) + 1) \times (\text{comp}(n_2) + 1) \\
&op \in (x^2) && \rightarrow && \text{comp}(n_1)^2 \\
&op \in (\sqrt[2]{x}) && \rightarrow && \text{comp}(n_1)^3 \\
&op \in (sin, cos, tan, exp, log) && \rightarrow && 2^{\text{comp}(n_1)}
\end{aligned}$$

3.4 Further Related Works

Some other works in GP worth mentioning include Age-Fitness Pareto Optimization, Multiple Regression GP, Multi-Gene GP, and the Evolutionary Demes Despeciation Algorithm.

Age-Fitness Pareto Optimization is inspired by the Age Layered Population Structure Algorithm (ALPS) and tries to combat premature convergence by introducing “age” of the individuals, the number of generations an individual has persisted in the population. The age is increased every generation or inherited by crossover and is used as a second objective criterion in a multi-objective optimization method [42].

Multi-Gene GP adds another level to the representation in GP by representing individuals as the weighted linear combination of several individual programs [43]. A similar approach is used in Multiple Regression GP in which each program is separated into its sub-expression, which are then recombined with a weighted linear combination using multiple regression [44].

The Evolutionary Demes Despeciation Algorithm (EDDA) is an initialization technique originally invented for the Geometric Semantic Operators introduced in Section 3.2 [45]. In this approach, the programs are initialized by taking the best individuals from a number of sub-populations, which have been evolved

independently by separate GP runs for several generations. By a certain percentage, these sub-populations are created using geometric semantic operators while the remaining populations use normal genetic operators.
An advancement to EDDA is the addition of using random sub-samples of the training data set for the creation of each separate sub-population [46].
The importance of the initialized population, or rather the randomness of the evolutionary process, has also been recognized in EA by using island population models, which are also used to run EA more efficiently in parallel. In these several separate sub-populations are evolved concurrently, which then transfer individuals between each other from time to time [11].

4 Implementation

One of the goals outlined in Section 1.1 is the comparison of the performance of GP against common regression methods. Some common regression methods include Polynomial Regression, Gradient Boosting, Neural Networks and Gaussian process regression. As a baseline model Linear regression is utilized. Comparing against Gaussian Process Regression is especially interesting as it is also currently used in the automotive industry in the Bosch ECU software, which is able to produce a model for the exhaust gas temperature [2].
Standard GP and two techniques described in Chapter 3, EPLEX and ParetoGP with the length and the Kommenda complexity of the programs as the second objective, are compared against the other regression methods.
The following chapter will explain the sources and the procedure to generate the training data and describe in detail how GP was implemented and give a short overview of the employed regression methods.

4.1 Training Data

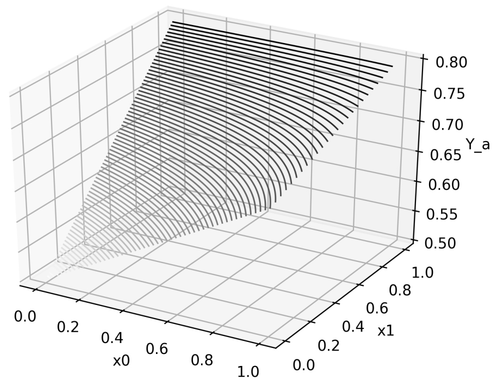
As mentioned in Section 1.1 two synthetic functions named f_1 and f_2 are used, which are displayed in Equations 4.1 and 4.2 respectively. Both functions have been supplied by IAV Powertrain Mechatronics Research and while they are synthetic in nature they are inspired by real-world data from automotive applications.

$$
\begin{aligned}
f_1 &= f_{1a} \cdot f_{1b} + f_{1c} \\
&\quad f_{1a} = 0.5 + 0.2 \cdot x_0 + 0.3 \cdot x_1 - 0.2 \cdot (x_0 \cdot x_1) \\
&\quad f_{1b} = 1 - 0.3 \cdot x_2 - 0.1 \cdot x_2^2 \\
&\quad f_{1c} = 0.1 - 0.1 \cdot (x_3 - 0.5)^2
\end{aligned} \tag{4.1}
$$

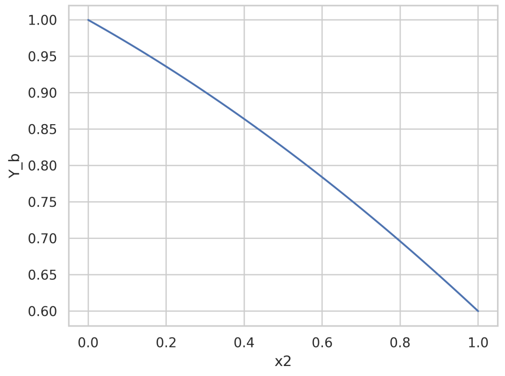
$$f_2 = \cos(2 \cdot x_0) + \sin(3 \cdot x_1) \tag{4.2}$$

Function f_1 has four input variables (x_0, x_1, x_2, x_3) while function f_2 has two input variables (x_0, x_1). All input variables are in the domain $[0, 1]$ and all functions have one output variable y.
Since function f_1 has more than three dimensions it can not be displayed in a simple plot without omitting important information, which is why its components f_{1a}, f_{1b}, f_{1c} and a histogram of 2000 output values of f_1 have been plotted instead in Figure 4.1.
While f_2 does contain the trigonometric functions sine and cosine the function f_1 does not and uses only the basic math operators in addition to the power of two.
A practical problem encountered with the function f_2 is that it is too simple and GP exactly learns the function in most cases or comes very close to it. One explanation is the syntax of the function, which is comparatively short and not very complex. The absence of any real numbers in f_2 also causes that the function can be displayed without including multiplications and constants, e.g. $2 \cdot x_0$ can be written as $x_0 + x_0$.
Because the results end up very similar in most cases it makes it difficult to compare different parameter settings. For that reason, another function named f_3 is used instead, which is an alternate version of f_2 with some added complexity. The function f_3 is displayed in Equation 4.3.

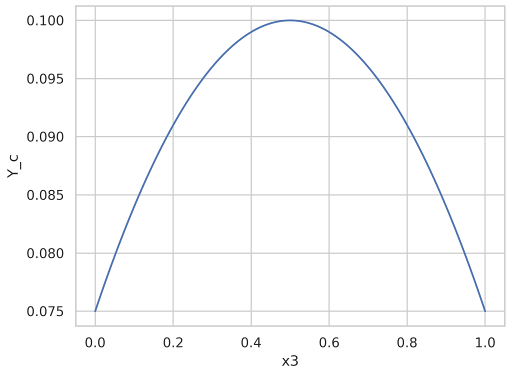
$$f_3 = 0.9 \cdot \cos(2.1 \cdot x_0) + 1.1 \cdot \sin(3.1 \cdot x_1) \tag{4.3}$$



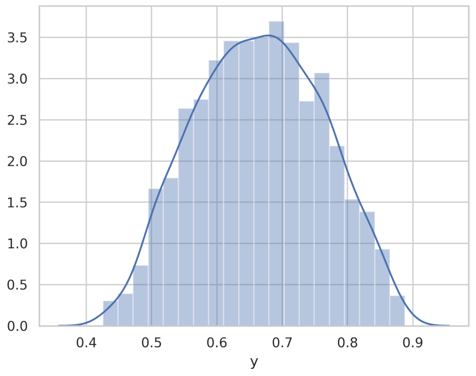
(a) Plot of f_{1a}



(b) Plot of f_{1b}



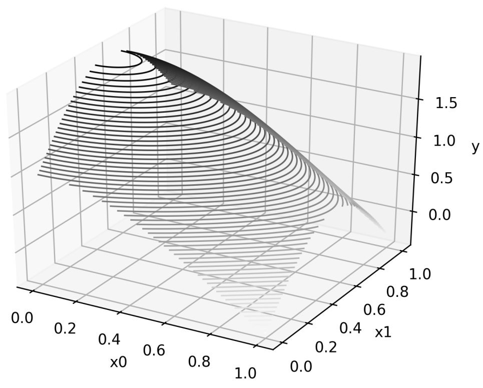
(c) Plot of f_{1c}



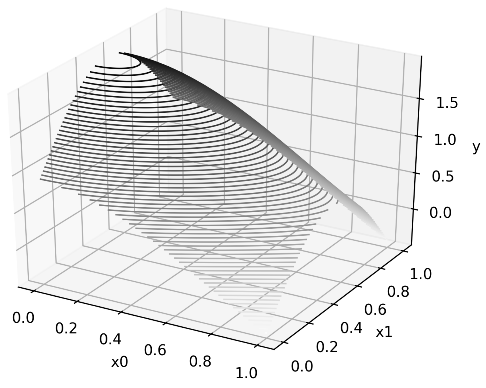
(d) Histogram plot of the output values of f_1 with a kernel density estimate

Figure 4.1: Plots showing the different components of f_1, including a density plot of f_1 from 2000 output values

The Figure 4.2 shows both f_2 and f_3 with their respective contour plots. The plots highlight that both functions are still very similar to each other with almost no visible differences existing. While both are still similar to each other the added complexity in f_3 should make it harder for GP to learn the function. Unfortunately, real-world data often contains noise from inaccuracies of the sensors and sometimes also contains errors because of wrongly handled data [47]. Therefore, GP should not only be able to learn the introduced synthetic functions but also provide robustness against noise in the data. Syn-



(a) Function f_2



(b) Function f_3

Figure 4.2: Plots of f_2 and f_3

thetic functions do not contain any noise or errors and to replicate the noise found in real-world data Equation 4.4 is employed.

$$f_{kn}^{\sigma} = f_k + \mathcal{N}(0, \frac{\sigma}{100}) \tag{4.4}$$

The noise is generated by adding random values from a normal Gaussian distribution with the mean at 0 to the output values. The specific noise versions are f_{kn}^1 and f_{kn}^5 of the functions f_1 and f_2. That is there are four additional functions f_{1n}^1, f_{1n}^5, f_{2n}^1, f_{2n}^5.
Since the noise adds sufficient complexity to the data, function f_2 was used instead of the alternate version f_3.

For each of the functions (f_1, f_{1n}^1, f_{1n}^5, f_2, f_3, f_{2n}^1, f_{2n}^5) a data set with 2000 instances is created, with the input values in the range $[0, 1]$. The data sets were generated in Python and saved in a CSV file with all values rounded to the 18th decimal.

4.2 GP Implementation

An existing implementation of a GP framework is used to save time on the implementation. That framework is then adapted to the needs outlined in this thesis, i.e. an implementation of ParetoGP, EPLEX and the Kommenda complexity.
Some of the existing frameworks that were looked into are ellenGP[1], Distributed Evolutionary Algorithms in Python (DEAP)[2] and gplearn[3].
The framework employed in this thesis is gplearn because it is a relatively simple and easy to understand implementation of GP, compared to ellenGP and DEAP, which are rather complex. EllenGP is written in C++ and DEAP is a framework for evolutionary algorithms in general and not only for GP. Gplearn is written in Python3 and also extends the scikit-learn machine learning library.
The implementation of GP in gplearn is mainly oriented by *A Field Guide to Genetic Programming* [12], that is standard GP with a tree-based representation as it was also introduced in Chapter 2. It also implements the initialization methods "grow", "full" and "half and half" as well as the basic genetic operators subtree crossover, subtree mutation, and point mutation.
Subtree mutation was implemented using the headless chicken method, i.e. subtree crossover with a randomly generated syntax-tree. In subtree crossover the crossover point is not chosen at uniform, instead, the terminals (leaves) in the tree have a 10% chance of being chosen while functions have a 90% chance. This is done to prevent the swapping of just leaves and to facilitate the swapping of more genetic material [5]. In this way, it is also done almost universally in the field of GP [48].
Gplearn uses a syntax tree-based representation, which is internally saved in prefix notation in a list datatype. Each function and terminal is one element in the list.
As the selection method tournament selection is employed and the fitness can be calculated with the MSE or the RMSE. For bloat prevention, the parsi-

[1]`https://epistasislab.github.io/ellyn/`
[2]`https://deap.readthedocs.io`
[3]`https://gplearn.readthedocs.io`

mony pressure method together with the length of individuals is implemented. The length is defined as the number of nodes in the syntax tree, or equivalent the number of elements in the prefix notation i.e. the list datatype.
The terminal set also contains ephemeral constants, which is another term for a random number generator for real numbers.
All math operators for the function set are implemented using the NumPy package for scientific computing in Python, which among others provides array objects and the possibility to perform calculations on entire arrays in one line of code [49]. By using the NumPy package all math operators in the gplearn implementation work with arrays as inputs as well as with just scalar values. This is an important functionality because the training data contains columns for each of the different input variables, which can be passed as arrays. Of the implemented math functions ($+$, $-$, $\times$, $\div$, $\sqrt[2]{x}$, log, $-x$, $\frac{1}{x}$, $|x|$, max, min, sin, cos, tan) the operators division, square root, logarithm and inverse have to be defined with protected versions (see Section 2.1).
The protected square root takes the square root of the absolute of a given value and the protected inverse uses the protected division operator to perform the calculation.
The protected division implemented in gplearn is the one recommended by Koza [5]. However, it is imprecise and does not reflect the behavior of division at the extremes very well. The implemented protected division returns 1 whenever the denominator is lower than 0.001, when the unprotected division operator would actually return higher values for lower denominators. The protected division was changed accordingly to perform the unprotected division until the absolute of the denominator is smaller than a specific ϵ-value (in this case ϵ =1e-10). If it is smaller the ϵ-value is added to the absolute of the denominator and the division is performed. The resulting value is then multiplied with -1 if the original denominator is negative to reverse taking the absolute of the denominator.
The protected logarithm was defined similarly to the protected division.
In Listing 4.1 the protected logarithm, the original and the new version of the protected division are displayed as pseudocode.
The Kommenda complexity from Definition 3.4 in Section 3.3.2 has exponen-

Listing 4.1: Protected Versions of Division and Log

```
EPS = 1e-10

old_protected_division(x1, x2):
   if abs(x2) > 0.001: return x1/x2
   else: return 1.0

new_protected_division(x1, x2):
   abs_x2 = abs(x2)
   if abs(x2) > EPS: return x1/x2
   else: return sign(x2) * x1/(abs_x2 + EPS)

protected_log(x1):
   abs_x1 = abs(x1)
   if abs_x1 > EPS: return log(abs_x1)
   else: return log(abs_x1 + EPS)
```

tial growth of the complexity value with the usage of the functions $\sqrt[2]{x}$, sin, cos, tan, exp, log (x^2 is not used in gplearn). The practice has shown that the growth frequently causes overflow errors (being in excess of 1e+308), resulting in complexities that are declared as $infinity$ in Python. For example, with the original Kommenda complexity an individual calculating 9 ($= \lceil \log_2 308 \rceil$) times the sine of a number larger than 5 would suffice to cause an overflow. This undesired behavior is prevented by adjusting the Kommenda complexity for the mentioned functions in a way that lessens the growth. The complexity for $\sqrt[2]{x}$ is redefined as $\text{comp}(n_1)^{1.15}$ while sin, cos, tan, exp, log are redefined as $\text{comp}(n_1)^{1.25}$.

EPLEX and ParetoGP are implemented as described in Section 3.3. The EPLEX version implemented is semi-dynamic EPLEX, because it is the one recommended by the authors of EPLEX.

One aspect of multi-objective optimization in GP is diversity control, which is mentioned briefly in Section 3.3. Kommenda et al. added to the diversity control by rounding the fitness of the individuals after a certain amount of decimals (essentially discretizing the fitness), which causes that very similar individuals are treated as equal [37]. This works well for the Pearson's R^2 employed by Kommenda et al., because it is in the range of [0,1] and the best

value is 1.0, i.e. the values are getting larger.
However, for fitness measures like the RMSE zero is the best value, i.e. the values will get smaller and have more leading zeros as decimals. This results in the ignored differences between individuals increasing, leading to a growing selection pressure until the rounding causes all individuals to have the same fitness of zero. Instead, the fitness values are rounded to the four significant digits. This is done by transforming the fitness values into the scientific notation (a float multiplied by a power of 10) and cutting off after three decimals. For example, the number $0,0987654$ is equivalent to $9,87654 * 10^{-2}$ and will then result in the value $9,876 * 10^{-2} = 0,09876$. This method has similarities to binning.
Parsimony pressure is turned off for ParetoGP, which leaves the size control solely in the responsibility of the ParetoGP algorithm. This only works to some extent because ParetoGP does not prevent very large programs but aims to build less complex versions. Moreover, it may lead to overfitting. Hence, another addition to ParetoGP is hard limits on the length of the programs for the ParetoGP archive [39]. There is also a limit on the minimum length to exclude very simple programs that have an extremely bad performance, which is detrimental to the evolutionary process, especially in the earlier generations.

4.3 Regression Methods

For a ready to use implementation of the regression methods the free software machine learning library Scikit-learn for the python programming language is employed [50].

Linear Multiple Regression

Linear Multiple Regression minimize the residual sum of squares of the errors (the least squares method) between the linear relationship of one or more

predictors variables (x_1,...,x_k) and one target variable y.
The general additive multiple regression model equation is [15]:

$$Y = \beta_0 + \beta_1 x_1 + ... + \beta_k x_k + \epsilon \tag{4.5}$$

Polynomial Regression

The polynomial regression employed is a special case of linear regression, which uses polynomial features of the variables on which the linear regression is then performed.

$$[x_1, x_2]_{\text{order}\equiv 2} \rightarrow [x_1, x_2, x_1 \cdot x_2, x_1^2, x_2^2] \tag{4.6}$$

In Equation 4.6 an example is displayed, in which the two input features x_1, x_2 are transformed into the second-order polynomials. The square brackets denote the set of features used for the linear regression. Instead of using the simple input features on the left side of the arrow the set on the right side is used. By doing this simple linear regression is extended to solve polynomial problems. For Polynomial Regression the input features can be transformed to polynomials of an arbitrarily chosen n-order.
In the experiments, the fourth-order polynomials are constructed for each data set.

Gradient Tree Boosting Regression

Gradient Tree Boosting is an ensemble method, which combines several weak learners (regression decision trees) into a strong learner. Weak learners are additively introduced to improve the model by using gradient descent on the error of a loss function [51].
In the experiments, the algorithm uses 400 estimators with a maximum depth of the trees set to 3.

Gaussian Process Regression

A Gaussian process describes a distribution over functions and is defined by a mean function and a covariance function, also called the kernel. In the regression process, the model is defined in a probabilistic way from a joint (multivariate) Gaussian distribution over a finite number of variables by maximizing the log-marginal-likelihood of the kernel parameters. From that model, a posterior predictive distribution can then be computed [52].
In the experiments, the kernel $1.0 \cdot \mathrm{RBF}(1.0)$ is used where RBF is the Radial Basis Function kernel. The optimizer used is the L-BFGS-B, the limited-memory Broyden–Fletcher–Goldfarb–Shanno algorithm with the extension to handle box constraints.

Multi-layer Perceptron

A multi-layer perceptron consists of an input layer, representing the input features, several hidden layers, and an output layer. Each hidden layer consists of neurons that are fully connected with the previous layer. In each neuron, the values from the previous layer are transformed into an output by using a weighted linear summation and a non-linear activation function, such as a rectifier or a hyperbolic function [11].
In Multi-layer Perceptron Regression (MLPR) a neural network is trained using backpropagation and no activation function is used in the output layer [50].
An optimal number of layers and number of neurons at each layer were tested beforehand. The MLP finally used in the experiments contains four hidden layers with 200 neurons each, as the activation function a rectified linear unit is employed and the solver used is the limited-memory Broyden–Fletcher–Goldfarb–Shanno (L-BFGS) algorithm. The MLP was learned with a maximum of 200 iterations with a tolerance at 1e-8 (training automatically stops when no improvement is shown in this range).

5 Evaluation

This chapter outlines and presents the experimental evaluation of the GP implementation introduced in Chapter 4. The central goal of the experiments is to answer the questions regarding the performance of GP, which are stated in Section 1.1:

- What parameter configurations for GP produce the best results?
- How does GP perform as a regression technique?
- How well does GP perform against established regressions methods?

The performance of GP is evaluated based on the result of the learning regarding the training data generated by the synthetic functions f_1 and f_3 introduced in Section 4.1. The result of the learning is then finally evaluated with the RMSE on the test data. In the evaluation the GP algorithms ParetoGP (PGP) with the length complexity, PGP with the kommenda complexity, EplexGP and StandardGP are used.

It is first evaluated what parameter settings produce the best results for the GP algorithms. In Section 5.2 experiments are first run to find the best parameter for the parsimony coefficient and in Section 5.3 the experiments are conducted and evaluated regarding the parameter of the crossover probability. The algorithms are compared against each other on the best parameter settings in more detail in Section 5.4. In Section 5.5 GP is also evaluated regarding its handling of noise in the training and test data (f^1_{1n}, f^5_{1n},f^1_{2n}, f^5_{2n}). Since the noise adds sufficient complexity to the data, the function f_2 was used instead of the alternate version f_3. In the following Section, it is evaluated how the results of GP regression compare to the results of the established regression methods (Linear, Polynomial, Gradient Tree Boosting, Gaussian Process, Multi-layer Perceptron) introduced in Section 4.3.

5.1 Experimental Setup

All possible parameter settings that are relevant for the GP Algorithm are shown in Table 5.1.

Table 5.1: GP Algorithm with possible parameter settings and the values they are set on. The last column contains a question mark if the value is not fixed and depends on the experiment being run.

Parameter	Value
Generations	500 | 1250
Population Size	800
ParetoGP lengths	Determined in Section 5.2
Selection	EPLEX | Tournament
Tournament Size	11
Elitism Size	1
Constants Range	(1e-10, 2)
Initialization Method	ramped half and half
Initialization Depths	(3, 6)
Function Set	$f_1, f_{1n}^1, f_{1n}^5 = \{+, -, \times, \div\}$
	$f_3, f_{2n}^1, f_{2n}^5 = \{+, -, \times, \div\ \sin, \cos\}$
Parsimony Coefficient	Determined in Section 5.2
P(C=Crossover)	Determined in Section 5.3
P(Subtree Mutation)	(1 - P(C)) / 2
P(Point Mutation)	(1 - P(C)) / 2
P(Point Replace)	0.05

The parameter *ParetoGP lengths* determines the minimum and maximum lengths of the programs that are allowed to be in the ParetoGP archive. The selection mechanism is determined by *Selection*, which is always used to draw programs from the population. That means Pareto optimization can also be used together with EPLEX, but is not used to together to evaluate them separately. *Constants Range* determines the range in which the ephemeral constants (random real numbers) are generated when being chosen from the terminal set. *Initialization Depth* is the minimum and maximum depth of the programs for the initialization method. The last four parameters are probabil-

ities for the different genetic operators. P(Point Replace) is for point mutation only and is the probability that any given node of a program is mutated.
The condition P(Crossover)+P(Subtree Mutation)+P(Point Mutation) ≤ 1.0 has to be satisfied. If the probabilities sum up to less than 1.0 the remaining probability (to reach a sum of 1.0) is used for a simple reproduction i.e. copying a program as is into the next population. However, in the experiments, this reproduction is avoided and to ensure the survival of the best program elitism is used instead. That is also why *Elitism Size* is set to 1, instead of a larger size.

Fixed parameter values were in part determined from small isolated experiments to be in an acceptable value range for good results.
Koza recommends a minimum population size of 500 and setting it as high as the GP system used can handle adequately [5]. The GP implementation in this thesis is limited by its usage of the main memory, which depends on the population size, the length of the programs and the number of jobs run in parallel. On a machine with 8 GB of RAM, a population size of 100,000 with one job in parallel could be run comfortably, although very slow in execution time. Academic literature regarding GP frequently uses population sizes arouind $1,000$, which is why the population size is set to 800 for the experiments [8, 34, 53].
An approximate good *Tournament Size* can be determined statistically beforehand. The probability of the best individual to have at least i descendants in the new population is equal to the probability of the best individual to be chosen because it always wins the tournament. That probability is described by the binomial distribution, with which it is possible to derive an indication for how large the size k of the tournament has to be to adequately represent the best individual in the new population. Generally, it is advisable to have the diversity in the population not affected negatively, i.e. have a low selection pressure [11]. For example, with a tournament size of 30, the expected number for the best individuals descendants is 18 (2.25% of a population of 800). However, a tournament size of 30 has the negative effect of converging after a few generations. The same is observed with a tournament size of 20. In decremental steps of 3 smaller tournament sizes were tested and a tournament

size of 11 has shown to produce a low convergence rate.
The *Initialization Depths* are commonly set to (2, 6) [5]. From the depths, it is possible to calculate the lengths (amount of nodes) of the programs to determine acceptable values. If it is hypothetically assumed that there are no unary functions in the function set all syntax trees of the programs are binary trees. The number of nodes in a binary tree with depth j is $2^j - 1$.
Programs with depth 2 have a length of 3 and programs with depth 6 the length 63. The synthetic functions f_1 and f_3 have the lengths 40 and 13, which means that syntax trees with a length of 3 do very poorly. An additional effect of choosing a higher minimum length is the introduction of more genetic material because the average amount of nodes per tree is higher. Larger programs also work better with the crossover operator, because there are more nodes beside the root and the leaves available as crossover points.
The minimum initialization depth is therefore set to 3 instead of 2, producing syntax trees with the minimum length of 7. A depth of 4 (length of 15) produces trees already too large.
The *Function Set* is adjusted depending on which synthetic function is currently used. For f_1 the function set $(+, -, \times, \div)$ is used and f_3 also includes the functions (sin, cos). f^1_{1n} and f^5_{1n} use the same function set as f_1 and f^1_{2n} and f^5_{2n} use the same function set as f_3. In a normal experiment setting without a priori knowledge about the synthetic functions that produce the training data experiments can be run to determine fitting function sets.

Before beginning to compare the different GP algorithms the best *Parsimony Coefficient* and the probabilities for the genetic operators are determined from experiments described in the following sections. In all experiments, each parameter setting or algorithm evaluated is run 31 times with different random seeds. Each run produces an RMSE from the best GP program on a training and test data set. Each data set contains 2000 instances, which are split 50:50 into a training and test data set. The training set is used for GP to learn, while the test data set is only used afterward for evaluation purposes. Of the 31 runs, the median is determined by the RMSE on the test data, which will then be taken to be compared to other parameter settings. Each run also records the length and Kommenda value of the best GP program produced.

The Mann–Whitney U test is used to determine if the differences in the results are significant. All tests are conducted pair-wise using the result with the best RMSE and are afterward corrected with the Holm-Bonferroni method for multiple comparisons. If the differences are significant a p-value smaller than 0.01 is produced.
The experiments for the parsimony parameter and the genetic operators are run for 500 generations, while the experiments to determine the best GP algorithm are run for 1250 generations.
The parameters for P(Subtree Mutation) and P(Point Mutation) are set to have the same value at all times, which is why they can be calculated automatically from the crossover parameter: $(1 - P(C))/2$. The rationale is that the experiments for the crossover are conducted to find differences between the crossover and mutation operators and not between the mutation operators. Another benefit is that the experiments do not have to be run for three different parameter configurations.
The parsimony coefficient and the probabilities for the genetic operators are not adapted for the synthetic functions with noise f^1_{1n}, f^5_{1n}, f^1_{2n}, f^5_{2n} to compare how GP handles noise with the settings for the noiseless versions. Adapting the parameters might also introduce a bias towards the noise. Additionally, it is unfeasible to also run the experiments for the noise versions because of the limited computational resources.

5.2 Parsimony Parameter

Table 5.2 shows the median and IQR of the RMSE on the test data and length for all examined parsimony coefficient values of the synthetic functions f_1 and f_3. The experiments were run by using StandardGP with length as complexity and tournament as selection; however, the parsimony coefficient is also used in EplexGP. The parameters for the genetic operators were left at the default values. StandardGP and EplexGP use the same parsimony coefficient to make the results more comparable to each other, the parsimony method is not used in ParetoGP. The parsimony coefficient is also heavily dependent on the problem GP is trying to learn, although the algorithm used

and other parameters that influence the length have an effect as well.
That the best parsimony coefficient is dependent on the kind of problem is reflected in the results in Table 5.2, in which f_3 has shown to produce the best results with a significantly lower parsimony coefficient than f_1.

Table 5.2: Median and IQR of the RMSE on the test data and Length for functions f_1 and f_3 using different parsimony values. RMSE and Length values marked with an asterisk are significantly different from the row in bold, which indicates the best RMSE, i.e. the lowest value.

	f_3				f_1			
Parsimony	RMSE		Length		RMSE		Length	
	Median	IQR	Median	IQR	Median	IQR	Median	IQR
7e-02	.3800 *	.0000	3 *	0.0	.0979 *	.0000	1 *	0.0
4e-02	.3800 *	.0000	3 *	0.0	.0979 *	.0000	1 *	0.0
1e-02	.0819 *	.2968	9 *	5.0	.0979 *	.0000	1 *	0.0
7e-03	.0621 *	.0842	13 *	5.5	.0979 *	.0000	1 *	0.0
4e-03	.0465 *	.0312	16 *	13.0	.0979 *	.0299	1 *	4.0
1e-03	.0372	.0268	41 *	19.0	.0381 *	.0742	13 *	19.0
7e-04	.0418	.0162	49 *	19.5	.0337 *	.0532	15 *	24.0
4e-04	.0310	.0253	64 *	21.5	.0255 *	.0575	27 *	36.0
1e-04	.0281	.0212	99	54.5	.0172 *	.0079	55 *	38.0
7e-05	**.0259**	**.0240**	**118**	**55.5**	.0168	.0066	63 *	43.0
4e-05	.0264	.0238	160	116.5	.0153	.0085	107 *	81.0
1e-05	.0287	.0250	238 *	155.0	.0127	.0095	163 *	107.0
7e-06	.0344	.0369	289 *	117.0	.0129	.0116	187	118.0
4e-06	.0389	.0264	395 *	224.5	**.0106**	**.0101**	**259**	**152.0**
1e-06	.0293	.0299	594 *	295.5	.0111	.0111	503 *	370.0

The results of the parsimony coefficients are further checked for significant differences using the Mann–Whitney U test with Holm–Bonferroni correction. The tests are conducted pair-wise with the best RMSE (see Table 5.2) for the values of the RMSE on the test data set and the length values. If a p-value smaller than 0.01 is produced the differences are significant.
Significant different values in Table 5.2 are marked with an asterisk. There are insignificant differences in the parsimony coefficient range of 1e-03 to 1e-06.

However, the test results on the length are significantly different outside of the range 1e-04 to 4e-05. Higher parsimony coefficient values automatically produce smaller programs, because they penalize larger programs more. The parsimony value 1e-03 is selected as the best value because it produces smaller programs while having an insignificant difference in the quality of the result (to the parsimony of 7e-05). The median length for 1e-03 is almost three times smaller than it is for 7e-05.

The test results for f_1 on the RMSE show insignificant differences in the range 7e-05 to 1e-05. Following the same argumentation as for f_3 the value 7e-05 is determined the best. The median length is four times smaller for 7e-05 compared to 4e-06, which is a more prominent difference than it is for the chosen parsimony for f_3.

Lower parsimony coefficients than 1e-06 (e.g. 7e-07) are not included in the experiments because values that low produce programs that have lengths in excess of 900. Programs of very large lengths frequently cause stack overflows in the parser of the GP implementation for some output values, such as the Kommenda complexity. Programs that large are also ineffective and too large programs are prone to overfitting, which is important to prevent especially for data in which noise is included.

The experiments for the parsimony coefficient also give an indication for the *ParetoGP lengths* parameter. An upper limit for the minimum length in *ParetoGP lengths* can be derived from the length of the synthetic functions used for the training of GP. Function f_1 has a length of 40 and f_3 a length of 13. However, programs that represent good sub-solutions should be accepted as well and usually, a priori knowledge about the perfect solution length is not available. It could be observed that programs with lengths smaller than 5 have a performance of very low value and can be discarded without repercussions.

To produce smaller programs the parsimony coefficients for f_1 and f_3 were chosen at the highest value that still resulted in non-significant values to the best median RMSE, while the idea in ParetoGP is that through Pareto optimization programs are automatically produced smaller. That reasoning is also why the maximum length can be chosen the same for f_3 as it is for f_1. The maximum length is generally maintained as a safety, to ensure the fast and

errorless execution of GP and to prevent overfitting [39].
The maximum should be set to something higher than what the chosen parsimony for f_1 (7e-05) produced as median length and can be set lower than the parsimony, which resulted in the best RMSE. The median length produced by the parsimony value between the best and the chosen parsimony for f_1 is selected as the maximum length allowed in the archive. The median length was rounded down, which results in 160.

5.3 Crossover and Mutation Parameters

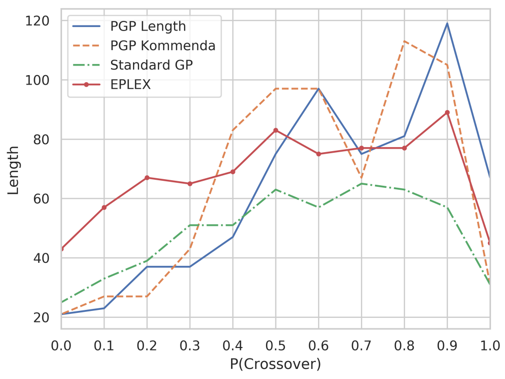
From the experiments for the parsimony parameter, it is concluded that the best parsimony coefficient values are 7e-05 for f_1 and 1e-03 for f_3, which are subsequently used to conduct the experiments for the crossover and mutation parameters.
Table 5.3 shows the median and IQR of the RMSE on the test data and the median length for the crossover probabilities from 1.0 to 0.0 in steps of 0.1 for functions f_1 and f_3. The RMSE values for f_1 are generally better than for f_3, the RMSE is however not suitable for comparisons across different test problems. For comparisons across test problems, the R^2 introduced in Section 2.3.1 is usually employed. A performance comparison is not the intention of this experiment.
The best crossover value seems to differ substantially across both test problems and the different GP algorithm combinations. The experiments' results are tested for significant differences using the Mann–Whitney U test with Holm–Bonferroni correction. This is done pair-wise for the RMSE on the test data set and the Length values using the values from the row with the best result, which is marked in bold. If a p-value smaller than 0.01 is produced the differences are significant. Significant different values in Table 5.3 are marked with an asterisk.The test results for f_1 in Table 5.3 surprisingly shows significant differences only in the lower and upper end of the crossover value spectrum.

Table 5.3: Median and IQR of the RMSE on the test data and Length for functions f_1 and f_3 using different crossover probabilities. RMSE and Length values marked with an asterisk are significantly different from the row in bold, which indicates the best RMSE, i.e. the lowest value. The last two rows only compare the crossover values 1.0 and 0.0 against each other.

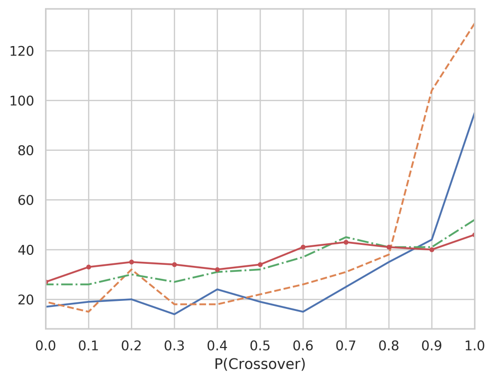
	PGP Length				PGP Kommenda				StandardGP				EplexGP			
Crossover	RMSE		Length		RMSE		Length		RMSE		Length		RMSE		Length	
	Median	IQR	Median	IQR	Median	IQR	Median	IQR	Median	IQR	Median	IQR	Median	IQR	Median	IQR
function f_1																
1.0	.0390 *	.0490	67	122.0	.0534 *	.0710	31	84.0	.0374 *	.0709	31	60.0	.0489 *	.0707	51 *	59.0
0.9	**.0132**	**.0119**	**119**	**101.0**	.0132	.0203	105	106.0	.0177	.0301	57	68.0	.0099	.0017	93	45.0
0.8	.0198	.0199	81	98.0	.0123	.0182	113	88.0	.0166	.0169	63	43.0	.0093	.0022	77	49.0
0.7	.0151	.0154	75	94.0	.0139	.0125	67	108.0	.0179	.0143	65	43.0	.0091	.0014	75	20.0
0.6	.0217	.0147	97	114.0	.0187	.0214	97	92.0	.0190	.0147	57	28.0	.0098	.0020	75	31.0
0.5	.0144	.0111	75	88.0	**.0118**	**.0087**	**97**	**96.0**	.0169	.0112	63	36.0	**.0089**	**.0017**	**83**	**46.0**
0.4	.0165	.0205	47	64.0	.0147	.0121	83	84.0	.0182	.0107	51	41.0	.0095	.0017	69	35.0
0.3	.0208	.0103	37	63.0	.0147	.0133	43	68.0	**.0159**	**.0134**	**51**	**28.0**	.0097	.0021	65	28.0
0.2	.0211	.0120	37 *	38.0	.0184	.0123	27 *	46.0	.0180	.0061	39	33.0	.0102	.0022	67	33.0
0.1	.0213 *	.0081	23 *	25.0	.0182 *	.0185	27 *	31.0	.0189	.0122	33	22.0	.0107 *	.0018	57 *	35.0
0.0	.0267	.0162	21 *	10.0	.0205 *	.0119	21 *	15.0	.0199	.0117	25 *	12.0	.0126 *	.0043	43 *	22.0
1.0	.0390 *	.0490	67	122.0	.0534 *	.0710	31	84.0	.0374 *	.0709	31	60.0	.0489 *	.0707	51	59.0
0.0	**.0267**	**.0162**	**21**	**10.0**	**.0205**	**.0119**	**21**	**15.0**	**.0199**	**.0117**	**25**	**12.0**	**.0126**	**.0043**	**43**	**22.0**
function f_3																
1.0	.0485 *	.0472	95 *	115.5	.0276	.0300	131	61.0	.0451 *	.0257	52 *	24.0	.0349	.0164	46	26.0
0.9	.0352	.0305	44	110.0	**.0263**	**.0168**	**104**	**110.5**	.0356	.0312	41	24.5	.0294	.0195	40	23.0
0.8	.0296	.0142	35	61.5	.0351	.0175	38	79.0	.0372	.0268	41	19.0	.0298	.0176	41	22.5
0.7	.0299	.0159	25	46.0	.0291	.0163	31	51.0	.0352	.0256	45	31.0	.0273	.0136	43	24.5
0.6	.0333	.0201	15	41.0	.0338	.0227	26	44.0	.0351	.0230	37	24.5	.0292	.0105	41	15.0
0.5	.0308	.0162	19	37.5	.0351	.0168	22 *	38.5	**.0281**	**.0148**	**32**	**27.5**	.0300	.0165	34	23.5
0.4	.0267	.0159	24	29.5	.0351	.0145	18 *	24.0	.0351	.0234	31	16.5	**.0253**	**.0136**	**32**	**11.0**
0.3	.0347	.0121	14	11.0	.0351	.0146	18 *	25.5	.0302	.0230	27	16.0	.0304	.0156	34	11.0
0.2	**.0220**	**.0158**	**20**	**20.5**	.0295	.0178	32	42.5	.0369	.0222	30	21.0	.0274	.0173	35	19.0
0.1	.0282	.0153	19	17.0	.0351	.0141	15 *	11.5	.0351	.0175	26	17.0	.0301	.0239	33	15.0
0.0	.0333	.0114	17	16.5	.0303	.0181	19 *	19.0	.0350	.0167	26	15.5	.0337	.0179	27	13.0
1.0	.0485 *	.0472	95 *	115.5	.0276	.0300	131 *	61.0	.0451	.0257	52 *	24.0	.0349	.0164	46 *	26.0
0.0	**.0333**	**.0114**	**17**	**16.5**	**.0303**	**.0181**	**19**	**19.0**	**.0350**	**.0167**	**26**	**15.5**	**.0337**	**.0179**	**27**	**13.0**

The tests for PGP Length and StandardGP even show insignificant differences for the crossover value 0.0 for the RMSE, although PGP Length shows significant differences for 0.1.
Nevertheless, the results of the test for PGP Kommenda and EplexGP show significant difference even for crossover with 0.0 and 0.1.
In general, the crossover value 0.0 produces better results than the crossover value 1.0. These results may convey that the mutation operators are more powerful and are more relevant for a successful GP run.
Across all tests results for f_3 in Table 5.3 there are only significant differences in the RMSE using PGP Length and StandardGP for crossover with 1.0. Similar to the results for f_1 this can also be interpreted as the mutation operators being more powerful.
The length values for f_3 on PGP Kommenda are significantly different in some cases, while they are in none for PGP Length besides for crossover 1.0. This can be explained by the fact that the significance test for PGP Kommenda was produced with the length value of the crossover with 0.9, which produces programs of larger length compared to the crossover with 0.2 in PGP Length.
Examining the hypothesis that either genetic operator is more powerful the crossover values 0.0 and 0.1 are tested for significant differences using the Mann–Whitney U test. The results are displayed in Table 5.3 in the last two rows for each function f_1 and f_3. For f_1 the difference between 0.0 and 1.0 are significant, while f_3 generally shows insignificant differences (excluding PGP Length). While it can be argued from the results, that the mutation operators are more powerful for f_1, the addition of the crossover operator has shown to produce significantly better results in some cases. The same can not be stated in case of function f_3, for which the results are ambiguous for both significance tests between crossover 1.0 and 0.0 and all crossover values.
An interesting result from the significance tests is that the differences in the lengths for f_1 are insignificant and for f_3 they are all significantly different. From Figure 5.1a it can be observed that for f_1 a higher crossover probability together with the mutation operators also produces larger programs. For f_3 the lengths differ less than they do for f_1. Surprisingly, the behavior of the crossover with 1.0 differs strongly between f_1 and f_3. Crossover with 1.0

produces smaller programs for f_1 than for crossover with 0.9 or 0.8, while for f_3 the crossover with 1.0 produces larger programs than for crossover with 0.9 or 0.8. This is especially the case for both PGP Length and PGP Kommenda. Subtree crossover causes growth in a program if the subtree it replaces is



(a) Lengths for f_1



(b) Lengths for f_3

Figure 5.1: Median lengths produced by different P(Crossover)

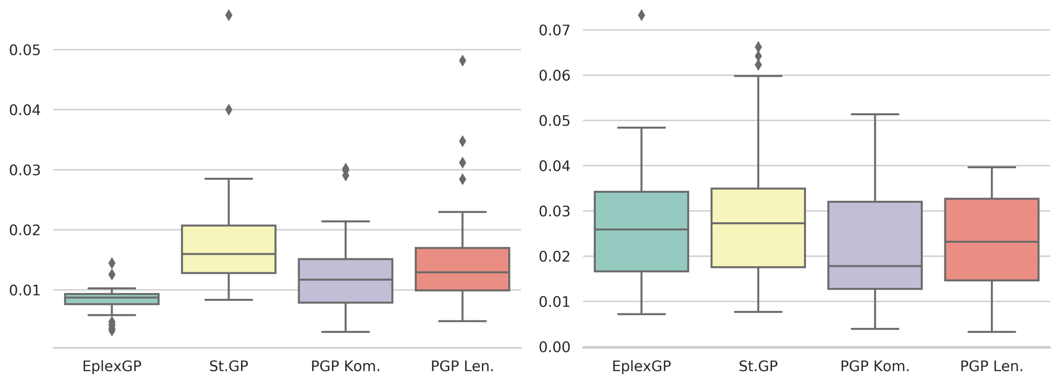
smaller than the one being inserted. However, subtree mutation works in the same way by choosing a crossover point. The difference is that the subtrees for subtree mutation are created randomly according to the parameter *Initialization Depths* and *Initialization Method*. That means on average the subtrees inserted by subtree mutation have the same length while the size of the subtrees inserted by subtree crossover depends on the size of the programs with the best performance. The best programs for f_1 tend to be larger programs than for f_3, which means the growth introduced by subtree crossover is larger as well. Additionally, subtree crossover only introduces growth into the population if these larger programs produce a better fitness.
For f_1 the programs may be smaller with a crossover at 1.0 because the crossover does not create better and larger programs, which is different for crossover with 0.9 in which new genetic material is introduced from the mutation operators, which can be utilized more effectively for the crossover to produce larger programs.
For f_3 the crossover with 1.0 produces larger programs because the effect of the mutation operators may be missing, which generally tend to produce programs of smaller size even if the crossover is included.

The effect of behaving differently using only crossover could also be caused by the fact that the perfect solutions are of different length, function f_1 has a length of 40 and f_3 a length of 13.
The crossover probability 0.5 is selected as the best for every algorithm and function combination, which is also a value frequently used in GP [12]. The value is chosen the same for each algorithm to make them more comparable against each other. This can also be done because the crossover value 0.5 does not show a significant difference for any algorithm. The value 0.5 produces the best results for f_1 using PGP Kommenda and EplexGP. The best value for f_1 using PGP Length is 0.9 and 0.5 does not produce significantly different results. The same is true using StandardGP, which has 0.3 as the best value. For f_3 only StandardGP produces the best results using a crossover probability of 0.5. However, PGP Length, PGP Kommenda, and EplexGP do not produce significantly different results when using crossover with 0.5. PGP Kommenda even produces significantly smaller results using a crossover probability of 0.5, compared to 0.9.

5.4 Best GP Algorithm

From the experiments for the parsimony and the crossover parameter, it is concluded that the best parsimony coefficient values are 7e-05 for f_1 and 1e-03 for f_3 and the best crossover parameter is 0.5, which are the values subsequently used to conduct the experiments for the best algorithm. The values used for the *ParetoGP lengths* parameter are (5, 160).
Varying from the previous experiments for the parsimony and crossover parameter the experiments for the best algorithm are run for 1250 generations each, instead of 500.
Figures 5.2a and 5.2b show boxplots of the RMSE on the test data for functions f_1 and f_3. The horizontal line inside each box is the median of the data, while the box represents the interquartile range (between the lowest 25% and the highest 75%). The whiskers at each end respectively depict 1.5 times the lower and upper quartile and every data point outside that range is

an outlier, which is plotted as a diamond shape.
For function f_1 EplexGP seems to be the clear winner and also is the algorithm with the lowest interquartile range, which means that it provides the best consistency. From all algorithms EplexGP also is the only one producing outliers outside the lower quartile, while all other algorithms only produce outliers outside the upper quartile. The interquartile ranges of the boxplots overlap only slightly, which displays how varied the results of the different algorithms are.



(a) for f_1 (b) for f_3

Figure 5.2: The results of the best algorithm experiment displayed as boxplots with the RMSE for the test data.

For function f_3 PGP Kommenda produces the best median RMSE on the test data. Generally, the interquartile ranges of the results overlap considerable and all algorithms seem to be producing results in the same intermediate range. The algorithms employing the parsimony method (EplexGP and Standard GP) do worse on function f_3 and also are the only ones producing outliers outside the upper quartile. The results could be connected to the fact that f_3 is considerable smaller than f_1, and thus harder to control with the parsimony method. Another explanation could be that the parsimony coefficient is not chosen optimally for f_3. ParetoGP seems to be able to adequately exploit its advantage of using the Pareto optimization principle by producing better results, instead on relying on the parsimony method.

ParetoGP Kommenda produces better results than ParetoGP Length for both functions.
In Table 5.4 the results of the experiment are displayed with the R^2 (coefficient of determination) instead of the RMSE making it possible to compare the results for the functions f_1 and f_3 against each other. The table displays the same results shown in the Figures 5.2a and 5.2b, but with the R^2 instead of the RMSE and the table also lists the median lengths produced by the algorithms and the IQR of the R^2 and the length.
To evaluate if the differences are significant, the results are tested for significant differences using the Mann–Whitney U test. This is done pair-wise for the R^2 and the Length with the values from the row containing the best R^2 on the test data set, which is marked in bold. If a p-value smaller than 0.01 is produced the differences are significant. Significant different values in Table 5.4 are marked with an asterisk.

Table 5.4: Median and IQR of the R^2 on the test data and Length for functions f_1 and f_3. R^2 and Length values marked with an asterisk are significantly different from the row in bold, which indicates the best R^2, i.e. the highest value.

	f_1				f_3			
Algorithm	R^2		Length		R^2		Length	
	Median	IQR	Median	IQR	Median	IQR	Median	IQR
EplexGP	**.9921**	**.0030**	**75**	**51.0**	.9977	.0031	37	21.5
StandardGP	.9735 *	.0274	61	30.0	.9974	.0032	21	26.5
PGP Kom.	.9858	.0173	125	68.0	**.9989**	**.0030**	**28**	**55.0**
PGP Length	.9827 *	.0198	113	64.0	.9981	.0030	23	26.0

For f_1 the only algorithm producing insignificant results compared to EplexGP is PGP Kommenda. The experiment for PGP Kommenda results in a median length almost double compared to the median length of EplexGP. However, which is not a significant difference according to the significance test.
For f_3 none of the algorithms produce significantly different results either for the R^2 or the length. With an R^2 in the range of 0.998 the GP algorithms also perform better for f_3 than for f_1. From Figures 5.2a and 5.2b it appears

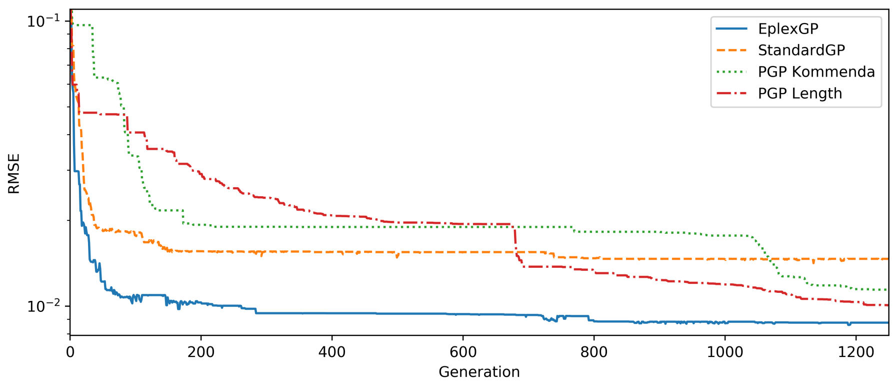
that the interquartile ranges for f_3 are generally larger than for f_1, because the RMSE is used which is not suited for comparisons across different data sets. Table 5.4 shows that the interquartile ranges for f_3 are actually smaller than they are for f_1. Better results and a smaller interquartile range might convey that it is easier for GP to learn f_3 than f_1, which is something that seems reasonable since function f_3 is smaller in length and less complex than f_1. That also means that f_1 being the harder function to learn is the better suited function to evaluate how well the GP algorithms perform.

The results for f_3 in Table 5.4 are also rather inconclusive in regard to what algorithm produces the better results, since all results have insignificant differences.

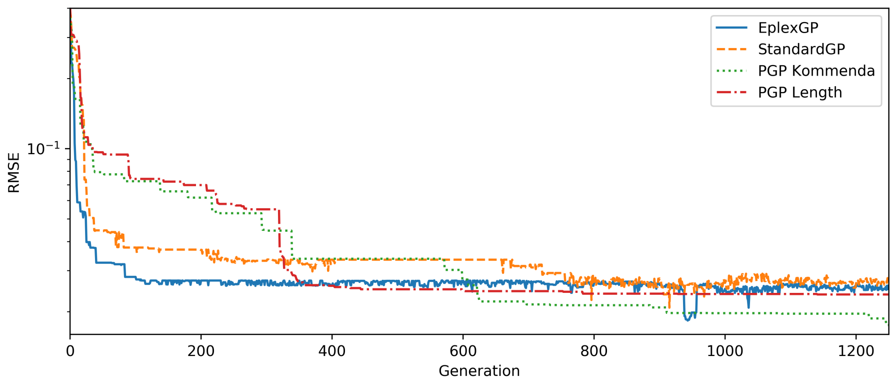
Something that is of interest in Evolutionary Algorithms is the convergence behavior of the algorithms. The runs producing the median RMSE on the test data are plotted over time (i.e. the generations) in Figures 5.3a and 5.3b to compare the convergence.

From Figure 5.3a it can be concluded that StandardGP and EplexGP converge rather early for f_1. For both algorithms this seems to be the case after about 100 to 200 generations with only minor improvements afterwards. The same can be said for f_3 in Figure 5.3b. For f_3 StandardGP and EplexGP also show oscillation in the performance in the later generations, which seems to occur because there is an "equilibrium" between the fitness and the length together with the parsimony coefficient. This results in individuals with a shorter length but a worse fitness being chosen over individuals with a longer length but a better fitness and vice versa, due to the penalty added by the parsimony method. This would support the earlier suspicion in the beginning of this section that the parsimony coefficient value for f_3 is not optimal or the parsimony method not being very suitable for small individuals.

PGP Length and PGP Kommenda create fitness plateaus for f_1, which only last for some generations and displaying major improvements in generations as late as 700 and 1100. For f_3 this is only true for PGP Kommenda, while PGP Length converges after about 400 generations. ParetoGP Length uses the length as the second optimization criterion and because it has a small discrete integer value range it allows for a lesser number of individuals in the archive



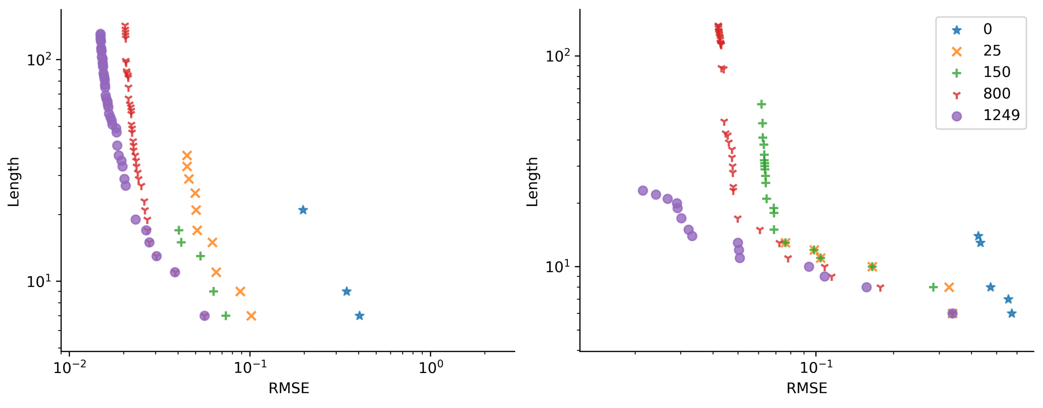
(a) for f_1



(b) for f_3

Figure 5.3: The runs producing the median RMSE plotted for each generation. The values plotted are the RMSE on the training data produced by the best individual from the current generation.

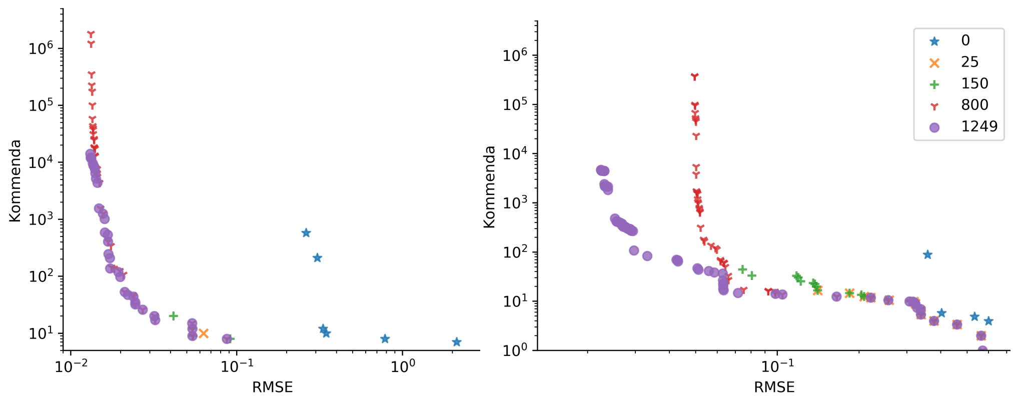
than PGP Kommenda does. Since f_3 generally requires shorter individuals, the number of individuals in the archive in PGP Length is automatically even smaller than for PGP Kommenda. For example, that means if the best individual has the length 20, all other individuals in the non-dominated archive of the ParetoGP algorithm have to have a worse fitness and a smaller length. In the example, the non-dominated archive would automatically be limited to a maximum number of 20 individuals since the lengths are always integers numbers. A new individual would then always have to beat an already existing individual in the archive to be included. This effect can also be observed in Figures 5.4a and 5.4b, in which the non-dominated archives of the ParetoGP algorithm from different generations of f_1 and f_3 are plotted. In both



(a) for f_1 (b) for f_3

Figure 5.4: Non-dominated archives of the ParetoGP algorithm for different generations for PGP Length.

Figures, a clustering behavior can be observed with each length in the upper limits represented each by one individual. This is especially apparent from Figure 5.4a in which the non-dominated archive in generation 1249 looks similar to a line plot. This behavior seems to be reduced when using a complexity measure such as Kommenda, which allows for greater diversity in the values it produces. This can be observed from the plotted non-dominated archives in Figures 5.5a and 5.5b. Another desirable effect from using the Kommenda



(a) for f_1 (b) for f_3

Figure 5.5: Non-dominated archives of the ParetoGP algorithm for different generations for PGP Kommenda.

complexity seems that instead of only producing individuals with a better fitness (see generation 800 and 1249 in Figure 5.4a), it is more likely to also produce individuals with a lower complexity (see generation 800 and 1249 in Figure 5.5a). The non-dominated archives in Figure 5.4a move only to the left, while in Figure 5.5a they move to the left and down.
For PGP Kommmenda the archives also overlap slightly, because some individuals are not replaced by better or shorter individuals, which is different from PGP Length, where there is almost no overlap between the archives.

5.5 Noise Robustness

The algorithms are run for the functions f^1_{1n}, f^5_{1n}, f^1_{2n}, f^5_{2n} to evaluate how robust they are towards noise in the data. The noise versions use the same settings as the noiseless versions of the functions.
From the experiments for the parsimony and the crossover parameter, it is concluded that the best parsimony coefficient values are 7e-05 for f_1 and 1e-03 for f_3 and the best crossover parameter is 0.5, which are the values subsequently used to conduct the experiments for the noise robustness. The values used for the *ParetoGP lengths* parameter are (5, 160).
Table 5.5 shows the median and IQR of the R^2 on the test and training data and the median length for the functions f^1_{1n}, f^5_{1n}. Table 5.6 shows the same for functions f^1_{2n}, f^5_{2n}. The experiments' results are also tested for significant differences using the Mann–Whitney U test.
In general, the algorithms are able to handle the function f^1_{1n} very well. Comparing the results on the training and test data set, the results are better on the test data set, which signifies that no overfitting seems to occur.
This changes when using f^5_{1n} for which the results are noticeable worse than for f^1_{1n}. The results on the test data set are also worse than on the training data set, which might indicate that the algorithms do overfit on the training data.
Comparing the median lengths of the results to the results for the noiseless functions in Table 5.4, the algorithms do not seem to compensate the noise with longer individuals.

Table 5.5: Median and IQR of the R^2 on the test and training data and length for functions f^1_{1n}, f^5_{1n}. R^2 and Length values marked with an asterisk are significantly different from the row in bold, which indicates the best R^2, i.e. the highest value.

Algorithm	R^2 on train		R^2 on test		Length	
	Median	IQR	Median	IQR	Median	IQR
	f^1_{1n}					
EplexGP	.9813	.0033	.9812	.0041	55 *	25.0
StandardGP	.9556 *	.0503	.9593 *	.0416	55 *	30.0
PGP Kom.	**.9818**	**.0157**	**.9834**	**.0138**	**105**	**77.0**
PGP Length	.9781	.0207	.9792	.0244	107	57.0
	f^5_{1n}					
EplexGP	.7874	.0038	.7829 *	.0058	33 *	10.0
StandardGP	.7789 *	.0192	.7627	.0354	45 *	27.0
PGP Kom.	.7939	.0165	.7777 *	.0276	113	98.0
PGP Length	**.7945**	**.0225**	**.7568**	**.1130**	**113**	**55.0**

Table 5.6: Median and IQR of the R^2 on the test and training data and length for functions f^1_{2n}, f^5_{2n}. R^2 and Length values marked with an asterisk are significantly different from the row in bold, which indicates the best R^2, i.e. the highest value.

Algorithm	R^2 on train		R^2 on test		Length	
	Median	IQR	Median	IQR	Median	IQR
	f^1_{2n}					
EplexGP	.9994	.0032	.9995	.0031	23	21.5
StandardGP	.9963 *	.0128	.9963	.0121	24	20.0
PGP Kom.	**.9996**	**.0054**	**.9996**	**.0050**	**14**	**11.0**
PGP Length	.9994	.0060	.9995	.0057	15	27.5
	f^5_{2n}					
EplexGP	.9900	.0027	.9907	.0028	17	13.5
StandardGP	.9880 *	.0097	.9888	.0106	18	15.5
PGP Kom.	**.9906**	**.0024**	**.9911**	**.0030**	**29**	**49.5**
PGP Length	.9905	.0017	.9911	.0026	15	42.0

The same can be said for the results of the functions f_{2n}^1, f_{2n}^5 in Table 5.6. However, some differences in the IQR of the lengths are noticeable.
The results on the functions f_{2n}^1, f_{2n}^5 are very surprising. They show no overfitting for either noise versions and the performance is very remarkable. As stated previously this is likely due to the fact that f_2 has a very low complexity, even with added noise.

5.6 GP vs. non-GP

Finally, the performance of GP is compared to the performance of well established regression methods. The general experiment settings for the non-GP methods are the same as for GP. That is, they are run 31 times with the same training and test data (a 50:50 split) with each separate run of the 31 runs done with a different random seed. Linear Regression and Polynomial Regression can not be run with a random seed. For Gaussian Process Regression the seed is used for the initialization of the centers. The RMSE results on the test data are displayed in Table 5.7 and also tested for significant differences. The significance tests are done pair-wise for the RMSE with the row containing the best RMSE, which is marked in bold. If a p-value smaller than 0.01 is produced the differences are significant. Significant different values in Table 5.7 are marked with an asterisk.
For f_1 and f_2 GP does noticeable worse than the best result. The Polynomial Regression method is even able to exactly learn the function with an error that can be considered zero. The function f_1 is a polynomial function with degree 4 and the Polynomial Regression method uses fourth-order polynomial features of the input values. Due to this, it is not very surprising that it is able to exactly learn the function. This is different for f_3 which is not a polynomial but uses sine and cosine functions.

Table 5.7: Median and IQR of the RMSE on the test data for functions f_1, f_3 from different machine learning methods. RMSE values marked with an asterisk are significantly different from the row in bold, which indicates the best RMSE, i.e. the lowest value.

Method	RMSE for f_1		RMSE for f_3	
	Median	IQR	Median	IQR
Linear	1.81e-02 *	0.00e+00	3.41e-01 *	0.00e+00
GradientTrees	6.99e-03 *	6.21e-06	2.78e-02 *	5.66e-05
MLP	1.66e-03 *	1.76e-04	5.29e-03 *	2.20e-03
Polynomial	**9.15e-16**	**0.00e+00**	4.94e-04 *	0.00e+00
GaussProcess	1.23e-06 *	0.00e+00	**8.67e-07**	**0.00e+00**
EplexGP	8.71e-03 *	1.51e-03	2.59e-02 *	1.76e-02
PGP Kommenda	1.17e-02 *	7.75e-03	1.78e-02 *	2.02e-02

For both functions, Gaussian Process Regression produces a very accurate model, which explains why it is currently used as the regression method in the Bosch ECU software [2].
The results of GP can still be regarded as good despite the worse performance when it is considered that GP produces interpretable analytical functions, which can give an insight into the general structure of the data used. This insight can then even be further used for other machine learning algorithms. The models of GP might also perform faster than the models of the other methods because they are more complex. GP also has the advantage that it automatically performs feature selection and might perform comparatively better to the other methods for data with higher dimensionality.
The Polynomial Regression method shares similar properties with GP because it also produces analytical functions. However, the Polynomial Regression method is bound to its model, the n-order polynomials created to learn the data and their linear relationship, which is why it does worse on function f_3 than on f_1. A problem in the Polynomial Regression method is also the number of polynomial features created, which scale exponentially with the degree. A too high degree can cause overfitting and also might make it suffer from the curse of dimensionality.
The best result on f_1 (RMSE ≈ 0.0029), which was produced by PGP Kom-

menda, is displayed as an example individual in a simplified version in Equation 5.1.

$$\begin{aligned}\widehat{f}_1 = {} & 0.0336 \cdot x_0 \cdot x_1 \cdot x_2 - 0.1831 \cdot x_0 \cdot x_1 - 0.0592 \cdot x_0 \cdot x_2 + 0.1938 \\ & \cdot x_0 - 0.0825 \cdot x_1 \cdot x_2 + 0.2908 \cdot x_1 - 0.0560 \cdot x_2^2 - 0.0112 \cdot x_2 \\ & \cdot x_3 - 0.1583 \cdot x_2 - 0.0224 \cdot x_3^3 - 0.0560 \cdot x_3^2 + 0.0780 \cdot x_3 + 0.5844\end{aligned} \tag{5.1}$$

The simplified version was generated by using SymPy[1], which is a Python library for symbolic mathematics. Additionally, the numbers are cut off after the fourth decimal to make the function easier to display and more readable. Surprisingly, cutting off the numbers did make the error on the data set better, instead of worse. This is a singular instance and could have the opposite effect on different individuals.

[1]`https://www.sympy.org`

6 Conclusion and Future Work

In this thesis, GP is introduced as a symbolic regression technique for usage in automotive modeling applications. One use case is the modeling of the exhaust temperature in vehicles, of which two representative synthetic functions were obtained to examine and evaluate the capabilities of Symbolic Regression Genetic Programming.
The current state-of-the-art in GP was extensively examined in Chapter 3. From these findings, suitable state-of-the-art techniques were used for an implementation of GP. The GP implementation created for this thesis was then further used to conduct experiments for the two synthetic problems.
The experiments indicate that the optimal value of the parsimony parameter is dependent on the problem used, which is likely related to the complexity and the size of the perfect solution for the problem.
The experiments for the crossover probability have shown that most settings for the probability of Crossover and Mutation do not produce significantly different results. However, a missing of either operator from GP will produce significantly worse results. Mutation produced better results than Crossover when evaluated separately without the other, which indicates that Mutation is more powerful. A higher probability for the Mutation Operator will also more likely result in shorter individuals. An equal division of the probabilities between both Operators was concluded as a good trade-off and deemed the best setting for GP.
EplexGP and PGP Kommenda produced the best results for the synthetic functions provided. EplexGP and StandardGP have shown to converge early, while PGP Kommenda and PGP Length displayed their superiority in using the Pareto optimization principle by not converging early. Using a more diverse complexity like Kommenda for ParetoGP produced better results.

While GP seems to be robust to a low noise level, higher amounts of noise in the data produce considerably worse results.
The results of GP compared to well established regression methods seem promising but also noticeably worse than for methods such as Gaussian Process Regression. While the results are noticeably worse than the methods compared to, GP produces an interpretable analytical function by using a quasi model-less method, which none of the methods compared to GP offer.
This thesis did not focus on producing the very best GP results possible but aimed at producing good and representative regression results for GP by using state-of-the-art methods. The potential of GP does not seem to be exhausted from the experiments in this thesis and it is within reason to conclude, that GP can produce better results than observed.

Future Work

Some parameter settings for GP were left at default or chosen from settings found in literature and not chosen experimentally to be the best setting possible. These include the Population Size, Elitism Size, Constants Range and the P(Point Replace) probability for the Point Mutation operator.
The experiments also did not examine the differences between Subtree Mutation and Point Mutation.
The GP representations introduced in Section 3.1 may produce different and better results than the syntax-tree-representation employed.
Several of the presented methods mentioned in the State-of-the-Art Chapter, which could improve the results of the symbolic regression process in GP even further, are not used and evaluated in this thesis due to time constraints or constraints regarding the computational resources.
This includes the Geometric Semantic Genetic Operators presented in Section 3.2, which have shown to produce better results on some polynomial functions and may produce better results.
The Evolutionary Demes Despeciation Algorithm (EDDA) mentioned in Section 3.4 is a technique originally developed for the Geometric Semantic Genetic Operators, but a similar approach would also prove useful for GP algorithms

that do not use the operators. The starting population in the GP implementation is created with the ramped half-and-half method and the individuals are generally of low quality. EDDA can improve the results in GP by creating a starting population with a high diversity and a high average fitness, which may also improve the large differences in fitness noticeable in separate runs of GP.
Island Population, also mentioned in Section 3.4, would help to maintain a high diversity in the population and prevent premature convergence in GP. It can also be used to run several different island population concurrently with different GP settings, which would render it unnecessary to decide on one exact good setting. Techniques to prevent premature convergence should be employed in GP in general.
As evident from the experiments in Chapter 5, Multi-Objective Selection in GP seems to be a very promising subject and should be a major focus of further research.
For the Polynomial Regression in the experiment, the fourth-order polynomial features of the input values are created to extend Linear Regression. This step can also be used as a pre-processing step for GP, which would remove the necessity for GP of creating them manually through the evolutionary process. Using this could make GP achieve similar or even better results than Polynomial Regression. This was not done due to time restrictions and to evaluate how GP performs without any kind of pre-processing of the data.

Bibliography

[1] Cristiano H. G. Brito, Cristiana B. Maia, and J. R. Sodré. "A Mathematical Model for the Exhaust Gas Temperature Profile of a Diesel Engine". In: *Journal of Physics: Conference Series* 633 (Sept. 2015), p. 012075. DOI: 10.1088/1742-6596/633/1/012075.

[2] Simon Wunderlin, A. Hurstel, and E. Kloppenburg. "Implementing a real time exhaust gas temperature model for a Diesel engine with ASC@ECU". In: *17. Internationales Stuttgarter Symposium.* Springer Fachmedien Wiesbaden, 2017, pp. 1041–1052. DOI: 10.1007/978-3-658-16988-6_79.

[3] Zheng Mao Ye and Habib Mohamadian. "Simple Engine Exhaust Temperature Modeling and System Identification Based on Markov Chain Monte Carlo". In: *Applied Mechanics and Materials* 598 (July 2014), pp. 224–228. DOI: 10.4028/www.scientific.net/amm.598.224.

[4] Zheng Mao Ye and Z.-J. Li. "Impact of lean-burn control technology on the fuel economy and NOx emission of gasoline engines". In: *Proceedings of the Institution of Mechanical Engineers, Part D: Journal of Automobile Engineering* 224.8 (May 2010), pp. 1041–1058. DOI: 10.1243/09544070jauto1409.

[5] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* Cambridge, MA, USA: MIT Press, 1992. ISBN: 0-262-11170-5.

[6] Leonardo Vanneschi et al. "Geometric Semantic Genetic Programming for Real Life Applications". In: *Genetic Programming Theory and Practice XI.* Ed. by Rick Riolo, Jason H. Moore, and Mark Kotanchek. New York, NY: Springer New York, 2014, pp. 191–209. DOI: 10.1007/978-1-4939-0375-7_11.

[7] Karolina Stanislawska, Krzysztof Krawiec, and Zbigniew W. Kundzewicz. "Modeling global temperature changes with genetic programming". In: *Computers and Mathematics with Applications* 64.12

(2012). Theory and Practice of Stochastic Modeling, pp. 3717–3728. DOI: 10.1016/j.camwa.2012.02.049.

[8] William La Cava, Lee Spector, and Kourosh Danai. "Epsilon-Lexicase Selection for Regression". In: *Proceedings of the Genetic and Evolutionary Computation Conference 2016*. GECCO '16. Denver, Colorado, USA: ACM, 2016, pp. 741–748. DOI: 10.1145/2908812.2908898.

[9] Akhil Garg, Ankit Garg, and K. Tai. "A multi-gene genetic programming model for estimating stress-dependent soil water retention curves". In: *Computational Geosciences* 18.1 (Feb. 2014), pp. 45–56. DOI: 10.1007/s10596-013-9381-z.

[10] William B. Langdon and Riccardo Poli. *Foundations of Genetic Programming*. Springer Berlin Heidelberg, 2002. DOI: 10.1007/978-3-662-04726-2.

[11] Rudolf Kruse et al. *Computational Intelligence*. Springer London, 2016. DOI: 10.1007/978-1-4471-7296-3.

[12] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A Field Guide to Genetic Programming*. Lulu Enterprises, UK Ltd, 2008. ISBN: 978-1-4092-0073-4.

[13] Wolfgang Banzhaf et al. *Genetic Programming: An Introduction: on the Automatic Evolution of Computer Programs and Its Applications*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998. ISBN: 1-55860-510-X.

[14] Tianfeng Chai and R R. Draxler. "Root mean square error (RMSE) or mean absolute error (MAE)?" In: *Geosci. Model Dev.* 7 (Jan. 2014). DOI: 10.5194/gmdd-7-1525-2014.

[15] Jay Devore. *Probability and statistics for engineering and the sciences*. Boston, MA: Brooks/Cole, Cengage Learning, 2012. ISBN: 978-0-538-73352-6.

[16] Markus F. Brameier and Wolfgang Banzhaf. *Linear Genetic Programming*. Boston, MA: Springer US, 2007. DOI: 10.1007/978-0-387-31030-5.

[17] Peter J. Angeline. "Subtree Crossover: Building Block Engine or Macromutation?" In: *Genetic Programming 1997: Proceedings of the Second Annual Conference*. Ed. by John R. Koza et al. Stanford University, CA, USA: Morgan Kaufmann, 13-16 7 1997, pp. 9–17. ISBN: 1-55860-483-9.

[18] Terence Soule, Rick L. Riolo, and Una-May O'Reilly. "Genetic Programming: Theory and Practice". In: *Genetic Programming Theory and Prac-*

tice VI. Boston, MA: Springer US, 2009, pp. 1–18. DOI: `10.1007/978-0-387-87623-8_1`.

[19] David R. White et al. “Better GP benchmarks: community survey results and proposals”. In: *Genetic Programming and Evolvable Machines* 14.1 (Dec. 2012), pp. 3–29. DOI: `10.1007/s10710-012-9177-2`.

[20] Lee Spector and Alan Robinson. “Genetic Programming and Autoconstructive Evolution with the Push Programming Language”. In: *Genetic Programming and Evolvable Machines* 3.1 (Mar. 2002), pp. 7–40. DOI: `10.1023/A:1014538503543`.

[21] William La Cava et al. “Genetic Programming with Epigenetic Local Search”. In: *Proceedings of the 2015 on Genetic and Evolutionary Computation Conference*. ACM Press, 2015. DOI: `10.1145/2739480.2754763`.

[22] Michael O’Neill and Conor Ryan. *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*. Springer US, 2003. DOI: `10.1007/978-1-4615-0447-4`.

[23] Evaldas Guogis and Alfonsas Misevičius. “Comparison of Genetic Programming, Grammatical Evolution and Gene Expression Programming Techniques”. In: *Information and Software Technologies*. Ed. by Giedre Dregvaite and Robertas Damasevicius. Cham: Springer International Publishing, 2014, pp. 182–193. ISBN: 978-3-319-11958-8.

[24] Julian F. Miller. “Cartesian Genetic Programming”. In: *Cartesian Genetic Programming*. Ed. by Julian F. Miller. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 17–34. DOI: `10.1007/978-3-642-17310-3_2`.

[25] Janet Clegg, James Walker, and Julian Miller. “A new crossover technique for Cartesian genetic programming”. In: Jan. 2007, pp. 1580–1587. DOI: `10.1145/1276958.1277276`.

[26] Simon L. Harding, Julian F. Miller, and Wolfgang Banzhaf. “Self-Modifying Cartesian Genetic Programming”. In: *Cartesian Genetic Programming*. Ed. by Julian F. Miller. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 101–124. DOI: `10.1007/978-3-642-17310-3_4`.

[27] William B. Langdon. “The evolution of size in variable length representations”. In: June 1998, pp. 633–638. DOI: `10.1109/ICEC.1998.700102`.

[28] Peter J. Angeline. “An investigation into the sensitivity of genetic programming to the frequency of leaf selection during subtree crossover”. In: *Genetic programming : proceedings of the first annual conference, 1996*. Cambridge, Mass: MIT Press, July 1996, pp. 21–29. ISBN: 0262611279.

[29] Lawrence Beadle and Colin G. Johnson. "Semantically driven crossover in genetic programming". In: *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*. IEEE, June 2008. DOI: `10.1109/cec.2008.4630784`.

[30] Lawrence Beadle and Colin G. Johnson. "Semantically driven mutation in genetic programming". In: *2009 IEEE Congress on Evolutionary Computation*. IEEE, May 2009. DOI: `10.1109/cec.2009.4983099`.

[31] Randal E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation". In: *IEEE Transactions on Computers* C-35.8 (Aug. 1986), pp. 677–691. DOI: `10.1109/tc.1986.1676819`.

[32] Alberto Moraglio, Krzysztof Krawiec, and Colin G. Johnson. "Geometric Semantic Genetic Programming". In: *Parallel Problem Solving from Nature - PPSN XII*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 21–31. DOI: `10.1007/978-3-642-32937-1_3`.

[33] Lee Spector. "Assessment of Problem Modality by Differential Performance of Lexicase Selection in Genetic Programming: A Preliminary Report". In: *Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation*. GECCO '12. Philadelphia, Pennsylvania, USA: ACM, 2012, pp. 401–408. DOI: `10.1145/2330784.2330846`.

[34] Thomas Helmuth, Lee Spector, and James Matheson. "Solving Uncompromising Problems With Lexicase Selection". In: *IEEE Transactions on Evolutionary Computation* 19.5 (Oct. 2015), pp. 630–643. DOI: `10.1109/tevc.2014.2362729`.

[35] William La Cava et al. "ϵ-Lexicase selection: a probabilistic and multi-objective analysis of lexicase selection in continuous domains". In: *CoRR* (2017). arXiv: `1709.05394v3`.

[36] Edwin D. De Jong and Jordan B. Pollack. "Multi-Objective Methods for Tree Size Control". In: *Genetic Programming and Evolvable Machines* 4.3 (Sept. 2003), pp. 211–233. DOI: `10.1023/A:1025122906870`.

[37] Michael Kommenda et al. "Evolving Simple Symbolic Regression Models by Multi-Objective Genetic Programming". In: *Genetic Programming Theory and Practice XIII*. Springer International Publishing, 2016, pp. 1–19. DOI: `10.1007/978-3-319-34223-8_1`.

[38] Stefan Bleuler et al. "Multiobjective Genetic Programming: Reducing Bloat Using SPEA2". In: *Proceedings of the IEEE Conference on Evolu-*

tionary Computation, ICEC 1 (May 2001). DOI: 10.1109/CEC.2001.934438.

[39] Guido F. Smits and Mark Kotanchek. "Pareto-Front Exploitation in Symbolic Regression". In: *Genetic Programming Theory and Practice II.* Springer-Verlag, pp. 283–299. DOI: 10.1007/0-387-23254-0_17.

[40] Ekaterina J. Vladislavleva, Guido .F. Smits, and Dick den Hertog. "Order of Nonlinearity as a Complexity Measure for Models Generated by Symbolic Regression via Pareto Genetic Programming". In: *IEEE Transactions on Evolutionary Computation* 13.2 (Apr. 2009), pp. 333–349. DOI: 10.1109/tevc.2008.926486.

[41] Theodore Rivlin. *Chebyshev polynomials : from approximation theory to algebra and number theory.* New York: Wiley, 1990. ISBN: 9780471628965.

[42] Michael Schmidt and Hod Lipson. "Age-Fitness Pareto Optimization". In: *Genetic Programming Theory and Practice VIII.* Ed. by Rick Riolo, Trent McConaghy, and Ekaterina Vladislavleva. New York, NY: Springer New York, 2011, pp. 129–146. DOI: 10.1007/978-1-4419-7747-2_8.

[43] Mark Hinchliffe et al. "Modelling Chemical Process Systems Using a Multi-Gene Genetic Programming Algorithm". In: *Late Breaking Papers at the Genetic Programming 1996 Conference Stanford University July 28-31, 1996.* Ed. by John R. Koza. Stanford University, CA, USA: Stanford Bookstore, 28–31 7 1996, pp. 56–65. ISBN: 0-18-201031-7.

[44] Ignacio Arnaldo, Krzysztof Krawiec, and Una-May O'Reilly. "Multiple Regression Genetic Programming". In: *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation.* GECCO '14. Vancouver, BC, Canada: ACM, 2014, pp. 879–886. DOI: 10.1145/2576768.2598291.

[45] Leonardo Vanneschi, Illya Bakurov, and Mauro Castelli. "An initialization technique for geometric semantic GP based on demes evolution and despeciation". In: *2017 IEEE Congress on Evolutionary Computation (CEC).* IEEE, June 2017. DOI: 10.1109/cec.2017.7969303.

[46] Illya Bakurov et al. "EDDA-V2 – An Improvement of the Evolutionary Demes Despeciation Algorithm". In: *Parallel Problem Solving from Nature – PPSN XV.* Ed. by Anne Auger et al. Cham: Springer International Publishing, 2018, pp. 185–196. ISBN: 978-3-319-99253-2.

[47] Xingquan Zhu and Xindong Wu. "Class Noise vs. Attribute Noise: A Quantitative Study". In: *Artificial Intelligence Review* 22.3 (Nov. 2004), pp. 177–210. DOI: 10.1007/s10462-004-0751-8.

[48] Stephen Dignum and Riccardo Poli. “Generalisation of the limiting distribution of program sizes in tree-based genetic programming and analysis of its effects on bloat”. In: Jan. 2007, pp. 1588–1595. DOI: 10.1145/1276958.1277277.

[49] Travis E. Oliphant. *NumPy: A guide to NumPy*. USA: Trelgol Publishing. 2006. URL: http://www.numpy.org.

[50] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830. URL: https://scikit-learn.org.

[51] Jerome H. Friedman. “Greedy function approximation: A gradient boosting machine.” In: *Ann. Statist.* 29.5 (Oct. 2001), pp. 1189–1232. DOI: 10.1214/aos/1013203451.

[52] Carl Rasmussen. *Gaussian processes for machine learning*. Cambridge, Mass: MIT Press, 2006. ISBN: 0-262-18253-X.

[53] Patryk Orzechowski, William La Cava, and Jason H. Moore. “Where are we now?” In: *Proceedings of the Genetic and Evolutionary Computation Conference on - GECCO ’18*. ACM Press, 2018. DOI: 10.1145/3205455.3205539.

Declaration of Authorship

I hereby declare that this thesis was created by me and me alone using only the stated sources and tools.

Hans-Martin Wulfmeyer   Magdeburg, July 17, 2019